André LaMothe

SAMS





TRICKS OF THE 3D GAME PROGRAMMING GURUS

ADVANCED 3D GRAPHICS AND RASTERIZATION

Foreword by Nolan Bushnell, Founder of Atari

Playrix Ex. 1023, page 1

00 0672318350 fm 05•02•2003 2:31 PM Page i

Tricks of the 3D Game Programming Gurus

Advanced 3D Graphics and Rasterization

André LaMothe

SAMS 201 West 103rd Street, Indianapolis, Indiana 46290

Playrix Ex. 1023, page 2

Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization

Copyright © 2003 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-31835-0

Library of Congress Catalog Card Number: 99-067392

Printed in the United States of America

First Printing: June 2003

06 05 04 03 4 3 2 1

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

1-317-581-3793

international@pearsontechgroup.com

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Associate Publisher
Michael Stephens

Acquisitions Editor
Kim Spilker

Development Editor
Mark Renfrow

Managing Editor
Charlotte Clapp

Project Editor
George E. Nedeff

Copy Editors
Seth Kerney
Mike Henry

Indexer
Erika Millen

Proofreaders
Kevin Ober
Suzanne Thomas

Technical Editor
David Franson

Team Coordinator
Cindy Teeters

Multimedia Developer
Dan Scherf

Graphic Designer
Gary Adair

Cover Illustrator
Alex Varanese

00 0672318350 fm 05•02•2003 2:31 PM Page iii

Contents at a Glance

Foreword xix
Introduction 1

Part I Introduction to 3D Game Programming

1 Getting Started with 3D Game Programming 9
2 A Crash Course in Windows and DirectX 69
3 3D Game Programming with a Virtual Computer 115

Part II 3D Math and Transformation

4 It's a Math, Math, Math World—Trigonometry, Vectors, Matrices, and Quaternions 241
5 Building a Math Engine 363
6 Introduction to 3D Graphics 493
7 Rendering Wireframe Worlds in 3D 621

Part III Basic 3D Rendering

8 Basic Lighting and Solid Modeling 743
9 Interpolative Shading Techniques and Affine Texture Mapping 869
10 Clipping in 3D++ 1005
11 Depth Buffering and Visibility 1087

Part IV Advanced 3D Rendering

12 Advanced Texturing Techniques 1147
13 Spatial Partitioning and Visible Algorithms 1307
14 Lighting, Shadows, and Secrets of the id 1439

Part V Advanced Animation and Physics Modeling and Optimization

15 Character Animation, Motion, and Collision Detection 1493
16 Optimization Technologies 1549

Playrix Ex. 1023, page 4

Part VI CD-ROM Appendixes

A About The CD-ROM

B Installing DirectX and Using Visual C/C++

C Trigonometry and Vector Reference

D C++ Primer

E Game Programming Resources

F ASCII Tables

Index 000

Table of Contents

Introduction 1

Part I Introduction to 3D Game Programming

1 Getting Started with 3D Game Programming 9

A Brief Introduction ... 9
The Elements of a 2D/3D Game ... 11
Section 1: Initialization ... 12
Section 2: Enter Game Loop ... 12
Section 3: Retrieve Player Input ... 12
Section 4: Perform AI and Game Logic ... 12
Section 5: Render Next Frame ... 13
Section 6: Synchronize Display ... 13
Section 7: Loop ... 13
Section 8: Shutdown ... 13
General Game Programming Guidelines ... 17
Using Tools ... 22
3D Level Editors ... 26
Setting Up to Get Down—Using The Compiler ... 28
A Sample 3D Game: `Raiders 3D` ... 32
The Event Loop ... 61
The Core 3D Game Logic ... 61
3D Projections ... 63
The Star Field ... 66
The Laser Cannons and Collision Detection ... 66
The Explosions ... 67
Playing `Raiders3D` ... 67
Summary ... 67

2 A Crash Course in Windows and DirectX 69

The Win32 Programming Model ... 69
The Bare Minimum for a Windows Program ... 71
It All Begins with `WinMain()` ... 71
A Basic Windows Application ... 77
The Windows Class ... 78
Registering the Windows Class ... 84
Creating the Window ... 85

00 0672318350 fm 05•02•2003 2:31 PM Page vi

The Event Handler ... 87
The Main Event Loop ... 94
Making a Real-Time Event Loop ... 99
DirectX and COM Crash Course ... 101
The HEL and HAL ... 103
The DirectX Foundation Classes in Depth ... 104
Flash Introduction to COM ... 106
What Exactly Is a COM Object? ... 107
Creating and Using DirectX COM Interfaces ... 110
Querying for Interfaces ... 111
Summary ... 114

3 3D Game Programming with a Virtual Computer 115

Introduction to the Virtual Computer Interface ... 115
Building the Virtual Computer Interface ... 118
The Frame Buffer and Video System ... 118
Working with Color ... 123
Animating the Display ... 125
The Complete Virtual Graphics System ... 128
I/O, Sound, and Music ... 129
The `T3DLIB` Game Console ... 130
`T3DLIB` System Overview ... 131
The Basic Game Console ... 131
The `T3DLIB1` Library ... 139
The DirectX Graphics Engine Architecture ... 139
Basic Definitions ... 140
Working Macros ... 142
Data Types and Structures ... 143
Function Prototypes ... 147
Global Domination ... 153
The DirectDraw Interface ... 155
2D Polygon Functions ... 160
Math and Error Functions ... 169
Bitmap Functions ... 171
8-Bit Palette Functions ... 177
Utility Functions ... 181
The BOB (Blitter Object) Engine ... 183
The `T3DLIB2` DirectX Input System ... 193
The `T3DLIB3` Sound and Music Library ... 201
The Header ... 201
The Types ... 202

Playrix Ex. 1023, page 7

00 0672318350 fm 05•02•2003 2:31 PM Page vii

Global Domination ... 203
The DirectSound API Wrapper ... 204
The DirectMusic API Rapper ... 210
Building the Final T3D Game Console ... 214
Mapping the Real Graphics to the Unreal Graphics of the Virtual Interface ... 214
The Final T3DLIB Game Console ... 218
Sample T3LIB Applications ... 231
Windowed Applications ... 231
Full-Screen Applications ... 232
Sound and Music ... 233
Working with Input ... 234
Summary ... 238

Part II 3D Math and Transformation

4 It's a Math, Math, Math World—Trigonometry, Vectors, Matrices, and Quaternions 241

Mathematical Notation ... 242
2D Coordinate Systems ... 243
2D Cartesian Coordinates ... 243
2D Polar Coordinates ... 246
3D Coordinate Systems ... 249
3D Cartesian Coordinates ... 249
3D Cylindrical Coordinates ... 252
Converting Between 3D Cartesian and Polar Coordinates ... 253
3D Spherical Coordinates ... 253
Trigonometry ... 256
The Right Triangle ... 256
Inverse Trigonometric Functions ... 259
Trigonometric Identities ... 259
Vectors ... 261
Vector Length ... 262
Normalization ... 263
Scalar Multiplication ... 263
Vector Addition ... 265
Vector Subtraction ... 265
The Inner Product, or the "Dot" Product ... 266
The Cross Product ... 270
The Zero Vector ... 272

Playrix Ex. 1023, page 8

00 0672318350 fm 05•02•2003 2:31 PM Page viii

Position and Displacement Vectors ... 272
Vectors as Linear Combinations ... 273
Matrices and Linear Algebra ... 273
The Identity Matrix ... 275
Matrix Addition ... 276
Transpose of a Matrix ... 276
Matrix Multiplication ... 277
Laws for Matrix Operations ... 279
Computing the Inverse and Solving Systems ... 279
Cramer's Rule ... 282
Transformations Using Matrices ... 285
Homogeneous Coordinates ... 286
Applied Matrix Transformations ... 287
Basic Geometric Entities ... 296
Points ... 297
Lines ... 297
Lines in 3D Space ... 300
Planes ... 302
Using Parametric Equations ... 307
2D and 3D Parametric Lines ... 307
Parametric Line Segments with Standard Direction Vector v ... 308
Parametric Line Segments with Unit Direction Vector |v| = 1 ... 309
Parametric Lines in 3D ... 311
Introduction to Quaternions ... 317
Complex Number Theory ... 318
Hyper-Complex Numbers ... 325
Applications of Quaternions ... 333
Basic Calculus ... 337
The Concept of Infinity ... 337
Limits ... 340
Summation and Finite Series ... 341
Infinite Series ... 344
Derivatives ... 345
Integrals ... 355
Summary ... 361

5 Building a Math Engine 363

Brief Overview of the Math Engine ... 364
File Structure of the Math Engine ... 364
Naming Conventions ... 365

Playrix Ex. 1023, page 9

00 0672318350 fm 05•02•2003 2:31 PM Page ix

Error Handling 366
My Final Word on C++ 366
Data Structures and Types 367
Vectors and Points 367
Parametric Lines 369
3D Planes 371
Matrices 372
Quaternions 376
Angular Coordinate System Support 377
Polar 2D 378
Cylindrical 3D 379
Spherical 3D 380
Fixed-Point Numbers 381
Math Constants 382
Macros and Inline Functions 384
General Utility and Conversion 390
Points and Vectors 390
Matrices 392
Quaternions 394
Fixed-Point Math 395
Prototypes 395
Globals 400
Math Engine API Listing 401
Trigonometric Functions 401
Coordinate System Support Functions 403
Vector Support Functions 407
Matrix Support Functions 418
2D and 3D Parametric Line Support Functions 434
3D Plane Support Functions 440
Quaternion Support Functions 446
Fixed-Point Math Support Functions 458
Equation Solving Support 464
Floating-Point Unit Math Primer 468
The FPU Architecture 469
The FPU Stack 470
The FPU Instruction Set 472
Classical Instruction Format 475
Memory Instruction Format 475
Register Instruction Format 476
Register Pop Instruction Format 477
FPU Examples 477

Playrix Ex. 1023, page 10

00 0672318350 fm 05•02•2003 2:31 PM Page x

FLD Examples ... 478
FST Examples ... 479
FADD Examples ... 480
FSUB Examples ... 484
FMUL Examples ... 485
FDIV Examples ... 487
Notes on Using the Math Engine ... 488
The Game Console ... 490
Comments on Math Optimization ... 491
Summary ... 491

6 Introduction to 3D Graphics 493

3D Engine Philosophy ... 494
The Structure of a 3D Game Engine ... 494
3D Engine ... 495
Game Engine ... 495
Input System and Networking ... 496
Animation System ... 496
Collision Detection and Navigation System ... 503
Physics Engine ... 503
Artificial Intelligence System ... 505
3D Model and Imagery Database ... 506
3D Coordinate Systems ... 508
Model Coordinates (Local Coordinates) ... 509
World Coordinates ... 513
Camera Coordinates ... 517
Final Word on Camera Coordinates ... 529
Hidden Object/Surface Removal and Clipping ... 531
Perspective Coordinates ... 539
The End of the Pipeline: Screen Coordinates ... 554
Basic 3D Data Structures ... 565
Considerations of Representing Polygonal Data in 3D ... 566
Defining Polygons ... 568
Defining Objects ... 576
Representing Worlds ... 581
3D Tools ... 583
Animation and Motion Data ... 584
Loading Data from the Outside World ... 585
PLG Files ... 585
NFF Files ... 589
3D Studio Files ... 594

Playrix Ex. 1023, page 11

00 0672318350 fm 05•02•2003 2:31 PM Page xi

Caligari COB Files ... 603
Microsoft DirectX .X Files ... 605
3D File Format Summary ... 606
Basic Rigid Transformations and Animation ... 606
3D Translation ... 606
3D Rotation ... 607
3D Morphing ... 610
Review of the Viewing Pipeline ... 612
Types of 3D Engines ... 614
Space Engines ... 614
Terrain Engines ... 615
FPS Indoor Engines ... 617
Ray Casters and Voxel Engines ... 618
Hybrid Engines ... 620
Integrating Everything into a Final Engine ... 620
Summary ... 620

7 Rendering Wireframe Worlds in 3D 621

General Wireframe Engine Architecture ... 622
The Data Structures and 3D Pipeline ... 623
The Master Polygon List ... 627
The New Software Modules ... 630
Writing a 3D File Loader ... 631
The .PLG/X File Loader ... 634
Building the 3D Pipeline ... 643
Generic Transformation Functions ... 643
The Local-to-World Transform ... 652
The Eulerian Camera Model ... 657
The UVN Camera Model ... 661
The World-to-Camera Transform ... 678
Object Culling ... 684
Back-face Removal ... 689
The Camera-to-Perspective Transform ... 694
The Perspective-to-Screen Viewport Transform ... 700
Hybrid Perspective Screen Transforms ... 707
Rendering a 3D World ... 711
3D Demos ... 716
Single 3D Triangle—DEMOII7_1.CPP¦EXE ... 717
3D Wireframe Cube—DEMOII7_2.CPP¦EXE ... 720
3D Wireframe Cube with Back-face Removal—
DEMOII7_3.CPP¦EXE ... 725

Playrix Ex. 1023, page 12

00 0672318350 fm 05•02•2003 2:31 PM Page xii

3D Tank Demo—DEMOII7_4.CPP¦EXE ... 727
3D Tanks Demo with Flying Camera—DEMOII7_5.CPP¦EXE ... 731
Battle Zone Walkthrough Demo—DEMOII7_6.CPP¦EXE ... 732
Summary ... 739

Part III Basic 3D Rendering

8 Basic Lighting and Solid Modeling 743

Basic Lighting Models for Computer Graphics ... 744
Color Models and Materials ... 747
Types of Lights ... 758
Lighting and Rasterizing Triangles ... 767
Getting Ready for Lighting ... 774
Defining Materials ... 775
Defining Lights ... 781
Shading in the Real World ... 788
16-Bit Shading ... 788
8-Bit Shading ... 789
A Robust RGB Model for 8-Bit Modes ... 789
A Simplified Intensity Model for 8-Bit Modes ... 795
Constant Shading ... 801
Flat Shading ... 804
Flat Shading in 8-bit Mode ... 825
Gouraud Shading Preview ... 828
Phong Shading Preview ... 831
Depth Sorting and the Painter's Algorithm ... 832
Working with New Model Formats ... 839
The Parser Class ... 839
Parsing Helper Functions ... 843
3D Studio MAX ASCII .ASC Format ... 847
trueSpace ASCII .COB Format ... 851
Quake II Binary .MD2 Format Preview ... 863
3D Modeling Tools Review ... 864
Summary ... 868

9 Interpolative Shading Techniques and Affine Texture Mapping 869

The New T3D Engine Features ... 870
Upgrading the T3D Data Structures and Design ... 871
New #defines ... 872
Added Math Structures ... 875
Utility Macros ... 877

Playrix Ex. 1023, page 13

00 0672318350 fm 05•02•2003 2:31 PM Page xiii

Adding Features to Represent 3D Mesh Data 878
Updating the Object Containers and Rendering List 887
Functional Listing Review and Prototypes 892
Rewriting the Object Loaders 901
Updating the Good Old .PLG/PLX Reader 902
Updating the 3D Studio .ASC Reader 919
Updating the Caligari .COB Reader 920
Polygon Rasterization Review 928
Triangle Rasterization 928
Fill Convention 934
Clipping 937
The New Triangle Functions 939
Optimizations 946
Implementing Gouraud Shading 948
Gouraud Shading Without Lighting 949
Adding Vertex Lighting for the Gouraud Shader 963
Basic Sampling Theory 976
Sampling in One Dimension 976
Bilinear Interpolation 980
Interpolating u and v 981
Implementing Affine Texture Mapping 985
Updating the Lighting/Rasterization Engine for Textures 988
Adding Lighting to 16-Bit Texture Rendering 989
Final Thoughts on Optimization Strategies for 8- and 16-Bit Modes 996
Lookup Tables 996
Mesh Vertex Coherence 997
Mathematical Caching 997
SIMD 998
Final Demos 999
Raiders 3D II 1000
Summary 1003

10 Clipping in 3D++ 1005

Introduction to Clipping 1005
Object Space Clipping 1005
Image Space Clipping 1010
Theoretical Coverage of Clipping Algorithms 1011
Clipping Fundamentals 1012
Cohen-Sutherland Clipping 1019
Cyrus-Beck/Liang-Barsky Clipping 1020

Playrix Ex. 1023, page 14

00 0672318350 fm 05•02•2003 2:31 PM Page xiv

Weiler-Atherton Clipping ... 1025
Further Studies on Clipping ... 1028
Practical View Frustrum Clipping ... 1029
The Geometry Pipeline and Data Structures ... 1031
Adding Clipping to the Engine ... 1032
A Little Fun with Terrain ... 1065
The Terrain Generating Function ... 1067
Generating the Terrain Data ... 1080
The Dune Buggy Demo ... 1080
Summary ... 1086

11 Depth Buffering and Visibility 1087

Introduction to Depth Buffering and Visibility ... 1087
Z-Buffering Basics ... 1091
Z-Buffer Problems ... 1093
Z-Buffer Example ... 1093
The Plane Equation Method ... 1097
Z Interpolation ... 1099
Z-Buffering Problems and 1/z Buffering ... 1101
A Sample Interpolation of Both z and 1/z ... 1103
Creating the Z-Buffer System ... 1106
Adding Z-Buffer Support to the Rasterizers ... 1111
Possible Z-Buffer Optimizations ... 1129
Using Less Memory ... 1129
Clearing the Z-Buffer Less Often ... 1129
Hybrid Z-Buffering ... 1131
Problems with the Z-Buffer ... 1132
The Software and Z-Buffer Demos ... 1132
Demo I: Z-Buffer Visualization ... 1133
Demo II: *Wave Raider* ... 1135
Summary ... 1143

Part IV Advanced 3D Rendering

12 Advanced Texturing Techniques 1147

Texturing—The Second Wave ... 1147
The Header File Data Structures ... 1149
Building a New Rasterizer Base ... 1160
The Final Decision to Go with Fixed-Point ... 1160
The New Non-Z-Buffered Rasterizers ... 1161
The New Z-Buffered Rasterizers ... 1165

Playrix Ex. 1023, page 15

00 0672318350 fm 05•02•2003 2:31 PM Page xv

Gouraud-Shaded Texturing 1167
Transparency and Alpha Blending 1177
Using Lookup Tables for Alpha Blending 1177
Supporting Alpha Blending at the Object Level 1194
Adding Alpha to the Terrain Generator for Fun 1203
Perspective-Correct Texturing and 1/z-Buffering 1207
Mathematical Foundations for Perspective Texture Mapping 1207
Adding 1/z-Buffering Support to the Rasterizers 1218
Implementing Perfect Perspective Mapping 1227
Implementing Linear Piecewise Perspective Mapping 1233
Exploring Quadratic Approximations for Perspective Texturing 1240
Optimizing Texturing with Hybrid Approaches 1248
Bilinear Texture Filtering 1250
Mip Mapping and Trilinear Texture Filtering 1257
Introduction to Fourier Analysis and Aliasing 1258
Creating the Mip Map Chain 1264
Mip Map Selection and Setup 1276
Trilinear Filtering 1284
Multiple Pass Rendering and Texturing 1285
Wrapping Things Up with a Single Call 1286
The New Rendering Context 1287
Setting Up the Rendering Context 1290
Making the Call to Render the Rendering Context 1293
Summary 1306

13 Spatial Partitioning and Visibility Algorithms 1307

The New Game Engine Module 1307
Introduction to Spatial Partitioning and Visible Surface Determination 1308
The Binary Space Partition 1313
Axis-Aligned Binary Space Partitioning 1314
Arbitrary Plane Space Partitioning 1315
Polygon-Aligned Space Partitioning 1316
Displaying/Visiting Each Node of a BSP Tree 1321
The BSP Tree Functions and Data Structures 1325
Creating the BSP Tree 1327
Splitting Strategies 1332
Traversing and Displaying the BSP Tree 1346
Integrating BSP Tree into the Graphics Pipeline 1360
The BSP Level Editor 1362
Limitations of the BSP 1375
Zero Overdraw Strategies Using BSP Trees 1376

Playrix Ex. 1023, page 16

00 0672318350 fm 05•02•2003 2:31 PM Page xvi

BSP Trees for Culling ... 1379
BSP Trees for Collision Detection ... 1391
Integrating BSP Trees and Standard Rendering ... 1392
Potentially Visible Sets ... 1399
Using the Potentially Visible Set ... 1402
Other Encodings of the Potentially Visible Set ... 1403
More Robust Computation of the PVS ... 1405
Portals ... 1408
Bounding Hierarchical Volumes and Octrees ... 1413
Using the BHV Tree ... 1415
Runtime Performance ... 1416
Selection Strategies ... 1418
Implementing BHV ... 1421
Octrees ... 1431
Occlusion Culling ... 1433
Occlusion Volumes ... 1433
Selection of Occluders ... 1436
Hybrid Occluder Selection ... 1436
Summary ... 1437

14 Shadows, Lighting, and Secrets of the id 1439

The New Game Engine Module ... 1439
Introduction and Game Plan ... 1440
The Simplified Physics of Shadows ... 1440
Ray Tracing Photons and Computing Intensity ... 1441
Simulating Shadows with Projective Images and Billboards ... 1445
Implementing Rasterizers with Transparency Support ... 1448
The New Software Library ... 1452
Simple Shadows ... 1454
Shadows that Scale ... 1458
Tracking the Light Source ... 1462
Final Thoughts on Simulated Shadows ... 1469
Planar Mesh Shadow Mapping ... 1470
Computing the Vector Transformations for Projection ... 1470
Optimizing Planar Shadows ... 1474
Introduction to Light Mapping and Surface Caching ... 1476
Surface Caching ... 1480
Generating the Light Maps ... 1481
Implementing the Light Mapper ... 1482
Dark Mapping ... 1486

Playrix Ex. 1023, page 17

00 0672318350 fm 05•02•2003 2:31 PM Page xvii

Special FX with Light Maps ... 1488
Optimizing the Light Mapping Code ... 1489
Putting Everything Together ... 1489
Summary ... 1489

Part V Advanced Animation, Physics Modeling, and Optimization

15 3D Character Animation, Motion, and Collision Detection 1493

The New Game Engine Module ... 1493
Introduction to 3D Animation ... 1494
The *Quake II* .MD2 File Format ... 1494
The .MD2 Header ... 1498
Loading *Quake II* .MD2 Files ... 1509
Animating .MD2 Files ... 1521
Simple Non-Character-Based Animation ... 1536
Rotational and Translational Motion ... 1536
Complex Parametric and Curved Motion ... 1539
Using Scripts for Motion ... 1540
3D Collision Detection ... 1543
Bounding Spheres and Cylinders ... 1544
Using Data Structures to Speed Collision Detection ... 1545
Terrain-Following Techniques ... 1546
Summary ... 1548

16 Optimization Technologies 1549

Introduction to Optimization Technologies ... 1550
Profiling Your Code with Microsoft Visual C++ and Intel VTune ... 1551
Profiling with Visual C++ ... 1551
Analyzing the Profile Data ... 1554
Optimizing with VTune ... 1556
Using the Intel C++ Compiler ... 1564
Downloading Intel's Optimizing Compiler ... 1565
Using the Intel Compiler ... 1565
Taking Advantage of Compiler Options ... 1566
Selecting Compiler Manually on a Source File Basis ... 1568
Optimization Strategies ... 1568
Single Instruction Multiple Data (SIMD) Programming Primer with SSE1569
Basic SIMD Architecture ... 1571
Working with SIMD in the Real World ... 1571
A SIMD 3D Vector Class ... 1586

Playrix Ex. 1023, page 18

General Optimization Tricks 1595
Trick 1: Getting Rid of `_ftol()` 1595
Trick 2: Setting the FPU Control Word 1595
Trick 3: Fast Zeroing of Floating-Point Values 1596
Trick 4: Fast Square Root 1597
Trick 5: Piecewise Linear Arctangent 1597
Trick 6: Pointer Increment 1598
Trick 7: Take ifs Out of Loops 1599
Trick 8: Branching the Pipeline 1600
Trick 9: Data Alignment 1600
Trick 10: Inline All Short Functions 1600
References 1600
Summary 1601

Part VI CD-ROM Appendixes

A About The CD-ROM

B Installing DirectX and Using Visual C/C++

C Trigonometry and Vector Reference

D C++ Primer

E Game Programming Resources

F ASCII Tables

Index xxx

Playrix Ex. 1023, page 19

Foreword

I am honored to be asked to write the foreword to this important work, teaching technical programmers the skills necessary for the creation of next-generation 3D video games from the ground up. There aren't many books that show you how to create a real-time 3D engine, starting with the plotting of a pixel. I reflect on how far the technology has come, and how primitive our games at Atari were. We were really pushing the state of the art then, but they really seem lame now.

To reminisce a bit, the early games were not even computer games in a technical sense. They were really just very fancy signal generators and state machines using counters and shift registers based on simple Boolean logic, built with nothing more than some slapped-together MSI (medium-scale integration) gates. You may remember that my first game (*Computer Space*) was created in 1970, four years *before* the Intel 4004, and six years before the 8080. I wish I had had a microprocessor to do them! Even then, the typical clock speed was much too slow for any significant real-time computing. The first game in which we used a microprocessor was *Asteroids*, and even then there was a great deal of hardware around it to augment the program, because the microprocessor couldn't do everything we needed in software.

Today, we march toward the capability to create images that are indistinguishable from a photograph (if we aren't already there). To create these images on the fly is truly a joy to behold. The realism and ability to create any world, environment, or character that these software and hardware tools give the game creator is a remarkable power. This palette of capability can shrink time, expand richness, and in general make new projects possible.

André LaMothe not only understands this technology intimately, but he also has an extraordinary game "sense." I have found over the years that many technologists are great practitioners of the art, but lack the sense of timing or tension that is necessary for a fun game. Others have a sense of gameplay, but are mediocre programmers. In André's case, you have both a true gamer and a software guru, and it comes through in every book he has written.

In a recent project that we did together, I was impressed not only with his knowledge of the state of the art, but also his extensive knowledge of the history of and intimacy with even some obscure games from the early days (ones that I thought only I remembered). Not to mention he wrote an entire game in 19 days for me! It is easy to remember the hits, but another thing to be knowledgeable about the stinkers (yes, Atari did a few that were really bad). But we did a lot that were great, and classics, as you know.

I hope you enjoy this book and use it as a launching pad for some great games that will keep me happily playing into the future.

Nolan Bushnell

Founder of Atari, Inc.

Playrix Ex. 1023, page 20

About the Author

André LaMothe has been involved in the computing industry and technology for more than a quarter century. He holds degrees in mathematics, computer science, and electrical engineering, and is one of the rare individuals that actually *did* work at NASA at the age of 20 doing research. His early teens and twenties were filled with consulting for numerous Silicon Valley companies, where he learned the realities of running a business and worked on his multidisciplinary background in fields such as telecommunications, virtual reality, robotics, compiler design, 3D engines, artificial intelligence, and other areas of computing and engineering.

His company Xtreme Games LLC was one of the first and last true "indie" publishers with a soul. Later he founded the Xtreme Games Developer Conference (XGDC) to give game developers a low-cost alternative to the GDC.

Lately he has been working on a number of projects, including eGamezone Networks, an online distribution system for games that's fair, fun, and has zero ads. Last but not least, he founded a new company, Nurve Networks LLC, to create handheld video game systems for value-minded consumers and hobbyists alike. Finally, he is the series editor for the world's largest game development series.

On a personal level, he likes everything extreme, from weightlifting, motorcycles, jet skis, and hotrods to "railing" on his blades. He has even trained extensively with the Shamrock Submission Fighting Team under the tutelage of Crazy Bob Cook, Frank Shamrock, and Javier Mendez. You probably don't want to get in an argument with him over DirectX or OpenGL—right or wrong, he will probably make you say uncle!

About the Technical Editor

David Franson has been a professional in the field of networking, programming, and 2D and 3D computer graphics since 1990. In 2000, he resigned his position as Information Systems Director of one of the largest entertainment law firms in New York City to pursue a full-time career in game development. He is also the author of *2D Artwork and 3D Modeling for Game Artists* (Premier Press), and is currently authoring *Xbox Hackz and Modz* (to be released June 2003).

Dedication

I dedicate this book to anyone with integrity—it's a lonely road, but someone has to walk it. Your actions do not go unnoticed…

Acknowledgments

First, I want to thank all the people at Sams Publishing. Let me make it clear that they have a very formulaic system for book publishing, and the *Tricks* series has always bent it way out of shape. We have to really applaud their ability to unplug from "Corporate America" once in a while and give everyone the latitude to create something that's a little special, so keep that in mind when reading their names below.

I would like to thank the publisher Michael Stephens, my acquisitions editor Kim Spilker, my trusty development editor Mark Renfrow, and of course, the project editor that keeps everything running smoothly, George Nedeff. There's also the nuts and bolts that go into a book, starting with the primary copy editor Seth Kerney, and of course, the technical editor David Franson (David also contributed some of the 3D models, including the Jet Ski, as well as many textures used in the demos).

Finally, no book would be complete without a CD, thanks to media developer Dan Scherf, and last but not least, the unsung hero that creates the index for the book—which is VERY important when you want to find something—anyway, thank you Erika Millen, for making a detailed and complete index for the book.

There's a special person that was part of this book when it first started the process that I don't want to go unmentioned; her name is Angela Kozlowski. She worked with me on the first 25% of the book, and really helped me define the overall "envelope" we were going to push; that is, "it's my way or the highway." She made everyone else understand how important and different this book would be. She never got to see the book finished because she moved on to another job, but now I am sure she will.

Next up are some of the companies and individuals that gave, loaned, or otherwise helped guide into my hands software and/or hardware. First is David Hackson of Intel Corporation for giving me copies of their latest advanced C++ and Fortran compiler products, as well as VTune. Then there was Kristine Gardner of Caligari Corporation, who gave me every version of trueSpace ever created. Long-time cohort Stacey Tsurusaki of Microsoft gave me that little inside edge at Microsoft. And of course, there are the companies that gave me evaluation copies of their products, or allowed me in one way or another to place

Playrix Ex. 1023, page 22

their software on the CD, such as JASC Inc. for the use of Paint Shop Pro, and Sonic Foundry for the use of Sound Forge, and Right Hemisphere's Mary Alice Krayecki for sending me copies of Deep Exploration and Deep Paint UV.

The next set of individuals are friends that dealt with me while I wrote this book; I know I am an "intense" person, but that's what makes me so much fun at parties!

Anyway, I would like to thank Mike Perone for helping get "software," and dealing with networking issues.

Mark Bell for listening to me complain—only another entrepreneur can understand the "hell" we go through—thanks Mark.

Sellam Ismail of the Vintage Computer Festival for getting me lots of cool "retro" stuff, and really cheap "junk."

John Romero for his frequent communications and giving me hope that at least someone in this business is having a little fun!

Also, to Nolan Bushnell for having me down to uWink Inc. and taking the time to talk to me about video games. It's not every day you get to spend time with the guy that invented Atari! And thanks for writing the foreword!

Also, Alex Varanese, my new apprentice, for putting up with my constant lecturing about perfection, deadlines, pushing the limits, visualizing success—okay, okay, I'll stop!

Last but not least, the people that really got the sharp point of my aggressive and demanding personality—mom, dad, and my patient girlfriend Anita and my puppy Ariel.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the *topic* of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Michael Stephens
Associate Publisher
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

For more information about this book or another Sams Publishing title, visit our Web site at `www.samspublishing.com`. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

01 0672318350 intro 05•02•2003 2:32 PM Page 1

Introduction

"Come get some…"

—Duke Nukem

Game Programming Principles and Practice

A long time ago, in a galaxy far, far, away, I wrote a book about game programming called *Tricks of the Windows Game Programming Gurus*. For me, it was an opportunity to create something that I had always wanted—a book that teaches readers how to make games. Anyway, it's been a few years, and I'm a little older, a little wiser, and I've definitely learned a lot of TRICKS <VBG>. You can get along fine without ever reading the first *Tricks*, but I warn you—this volume is definitely more advanced, focuses on 3D, and assumes a lot.

With that in mind, this book is going to continue where the first *Tricks* left off, and further continue 3D concepts. I'm going to cover every major topic in 3D game programming that I can fit within the bounds of time and space.

Now, I'm not going to assume that you are a master programmer, or that you even know how to make games. However, this book is for not for beginners, and is really targeted toward intermediate to advanced game programmers, or intermediate to advanced programmers in general. If you don't know how to program in C/C++, you're going to be lost! So, newbies beware! There are many great books on C/C++, and I suggest anything by Stephen Prata or Robert Lafore—they are the best C/C++ authors in the world as far as I am concerned.

Today is probably the coolest time in history to be in the game business. We now have the technology to create games that do look real! Imagine what will come next—and you think the PlayStation 2, Xbox, and GameCube are cool! But all this technology isn't easy or trivial to understand—it takes really hard work and lots of Red Bull.

The difficulty level of game programming has definitely been cranked up these days in relation to the skill set needed to make games. But if you're reading this, you are probably one of those people that likes a challenge, right? Well, you came to the right place, because when you're done with this book, you will be able to create a full 3D, texture-mapped, fully-lit video game for the PC with a software rasterizer you write yourself. Moreover, you will understand the underlying principles of 3D graphics and be able to better understand and utilize 3D hardware today and in the future.

Playrix Ex. 1023, page 26

01 0672318350 intro 05•02•2003 2:32 PM Page 2

What You're Going to Learn

In this book, you're going to learn about 100^{10} teraquads of information! I'm going to fill your neural net so full of info, you'll have synaptic leakage! Seriously though, this book covers all the elements necessary to create a Windows 9x/2000-based game for the PC, including such topics as:

- A basic understanding of the engine from the first *Tricks*
- Win32 programming and DirectX Foundation 7.0+
- Advanced mathematics, including quaternions
- 2D/3D graphics and algorithms
- 3D projections and camera manipulation
- Wireframe and solid rendering
- Lighting and texture mapping
- Advanced visibility algorithms
- 3D animation techniques

And a lot more…

The first thing that you might be wondering about is, does this book focus on using 3D hardware, or does it show how to actually implement 3D software rasterizers with code? Well, the answer is software, primarily (with a "but" that I will get to in a minute)! Only wimps rely on 3D hardware. A real game programmer can write a 3D engine from scratch with blood pouring out of his eyes, on fire, and with needles in his eardrums, as well as use 3D hardware, if available. So, we're going to focus on real 3D programming in this book. With that knowledge, you can learn any 3D API in a couple of weeks.

My philosophy is: If you can understand how to write a texture mapper or a viewing system, you will have that much more of an advantage in using hardware. Furthermore, it will be some time before every computer has *good* 3D hardware, so you can't assume it exists, and it would be a bad thing to cut out all those computers from your target market just because they don't have 3D hardware (the Midwest and Europe are perfect examples of geographic areas that don't have the latest and greatest). Especially if you want to get into the game market without millions of dollars—you will start in the value software market, which is primarily non-3D accelerated. Finally, it's just more fun!

Finally, I'm sure you are a little worried about this whole "Windows-DirectX" deal. Well, with the right approach, Windows programming is actually very simple, fun, and takes care of a lot of problems that we had with DOS32. Don't think of Windows programming as a problem—think of it as an opportunity to spend more time on game code, rather than trivial details like GUI stuff, I/O, and graphics drivers. Believe me, you would spend 24 hours a day if you had to write graphics drivers for every 2D/3D accelerator that came on the market. Then there are sound cards, joysticks, blah, blah, blah…

Playrix Ex. 1023, page 27

01 0672318350 intro 05•02•2003 2:32 PM Page 3

What You Absolutely Need to Know

This book assumes that you can program, and program pretty well. You're going to be fairly lost if you can't write C code and don't know how to use a compiler in your sleep. However, the book uses some C++—enough to make a C coder just a little uneasy. I'll warn you if I'm doing anything weird. Also, there's a decent C++ primer in Appendix D, so check it out if you need a crash course. Basically, C++ is only needed here and there for examples using DirectX.

Nevertheless, I've decided that I'm going to use C++ a little more on this book because there are so many things in game programming that are object-oriented, and it's sacrilege to force them to be C-like structures. Bottom line: If you can program in C, you should be fine. If you program in C/C++, you shouldn't trip out at all.

Everyone knows that a computer program is nothing more than logic and math. Well, 3D video games have the emphasis on the math part! 3D graphics is all math. However, luckily for us, it's cool math! Yes, math can be cool. About the only thing you need to know is basic algebra and geometry. The vector, matrix, and calculus stuff I'll teach you along the way. Heck, if you can add, subtract, multiply, and divide, you will still be able to understand 90% of what's going on, even though you might not be able to re-derive it. But as long as you can use the code, that's all the matters in the end. And that Carmen Electra is single again <GRIN>.

That's really all you need to know. Of course, you'd better call all your friends and tell them that they won't see you for about two years because you're going to be a little busy. But just think of all the movies you'll get to rent when you're done with your training!

How This Book Is Organized

This book is separated into six major parts:

- Part I: Introduction to 3D Game Programming—This section introduces you to game programming, Windows, and DirectX, and we build up the virtual computer interface used to create all the demos.
- Part II: 3D Math and Transformation—This section introduces all the mathematical concepts and builds up a complete mathematics library for use in the book. The latter parts of the section cover 3D graphics, data structures, cameras, and wireframe rendering.
- Part III: Basic 3D Rendering—This section covers lighting, basic shading, and hidden surface removal. Full 3D clipping is also covered.
- Part IV: Advanced 3D Rendering—This section covers texture mapping, advanced lighting, shadows, and spatial partitioning algorithms, such as BSP trees, portals, and more.

Playrix Ex. 1023, page 28

01 0672318350 intro 05•02•2003 2:32 PM Page 4

- Part V: Advanced Animation, Physics Modeling, and Optimization—This section covers animation, motion, collision detection, and simple physics modeling. Hierarchical modeling is also covered, along with loading large worlds. We'll also cover numerous optimization techniques.

Installing the CD-ROM

The CD-ROM contains all the source, executables, sample programs, stock art, 3D modelers, sound effects, and bonus technical articles that make up the book. Here's the directory structure:

```
T3DIIGAME\

SOURCE\
       T3DIICHAP01\
       T3DIICHAP02\
                .
                .
       T3DIICHAP16\

TOOLS\
GAMES\
MEDIA\
        BITMAPS\
        3DMODELS\
        SOUND\


DIRECTX\


ARTICLES\
```

Each main directory contains specific data that you'll need. Here's a more detailed breakdown:

- `T3DGAMEII`—The root directory that contains all other directories. Be sure to read the `README.TXT` within it for any last-minute changes.
- `SOURCE`—Contains all the source directories for the book in chapter order. Simply drag the entire `SOURCE\` directory to your hard drive and work from there.
- `MEDIA`—Contains stock 2D artwork,3D models, and sounds that you may use royalty-free in your games.
- `DIRECTX`—Contains the latest version of the DirectX SDK.

Playrix Ex. 1023, page 29

01 0672318350 intro 05•02•2003 2:32 PM Page 5

- GAMES—Contains a number of 2D and 3D shareware games that I think are cool!
- ARTICLES—Contains articles written by various experts in the field of game programming for your edification.
- TOOLS—Contains various applications and tools you might find useful. Most can be downloaded from internet, but they are rather large, so these versions will save you the download time.

There isn't any general installation program for the CD because there are so many different types of programs and data. I leave the installation to you. However, in most cases, you'll simply copy the SOURCE\ directory to your hard drive and work within it. As for the other programs and data, you'll probably install them as you need them.

Installing DirectX

About the only important part of the CD that you must install is the DirectX SDK and runtime files. The installation program is located within the DIRECTX\ directory, along with a README.TXT file explaining any last-minute changes.

NOTE

You must install the DirectX 8.1 SDK or better to work with demos and source from this book (DirectX 9.0 is on the CD though), so if you're not sure that you have the latest files on your system, run the installation and it will tell you.

Compiling the Programs

I wrote the programs in this book with Microsoft Visual C++ 6.0. However, the programs will work with any Win32-compliant compiler in most cases. Nevertheless, I suggest Microsoft Visual C++ or .NET, because they work the best for this type of work.

If you are unfamiliar with your compiler's IDE, you are going to be wildly lost when compiling Windows programs. Please take the time to learn your way around the compiler, and at least how to compile a "Hello World" console program or something before you dive into compiling the programs.

To compile Windows Win32 .EXE programs, all you need to do is set the target of your program project to Win32 .EXE and compile. However, to create DirectX programs, you must include the DirectX import libraries in your project. You might think that you can simply add the DirectX libraries to your include path, but this won't work! Save yourself a headache and include the DirectX .LIB files in your project or workspace manually. You can find the .LIB files in the LIB\ directory, right under the main DirectX SDK directory that you installed under. This way, there won't be any linker confusion. In most cases, you'll need the following:

Playrix Ex. 1023, page 30

01 0672318350 intro 05•02•2003 2:32 PM Page 6

- `DDRAW.LIB`—DirectDraw import library
- `DINPUT.LIB`—DirectInput import library
- `DINPUT8.LIB`—DirectInput import library
- `DSOUND.LIB`—DirectSound import library
- `WINMM.LIB`—Windows Multimedia Extensions

I'll go into more detail on these files when you actually start working with them, but at least keep them in mind when you start getting "unresolved symbol" errors from your linker. I don't want any emails on this subject from rookies!

In addition to the DirectX .LIB files, you must include the DirectX .H header files in your header search path, so keep that in mind. Also, make sure to make the DirectX SDK directories first in the search path list, because many C++ compilers have old versions of DirectX within them, and the old headers might be found in the compiler's own `INCLUDE\`, which is wrong. The proper place is the DirectX SDK include directory, which is located in the main installation directory of the DirectX SDK in `INCLUDE\`.

Finally, for those of you that use Borland products, make sure that you use the Borland versions of the DirectX .LIB files. They can be found in the `BORLAND\` directory of the DirectX SDK installation.

Playrix Ex. 1023, page 31

02 0672318350 Part1 05•02•2003 2:32 PM Page 7

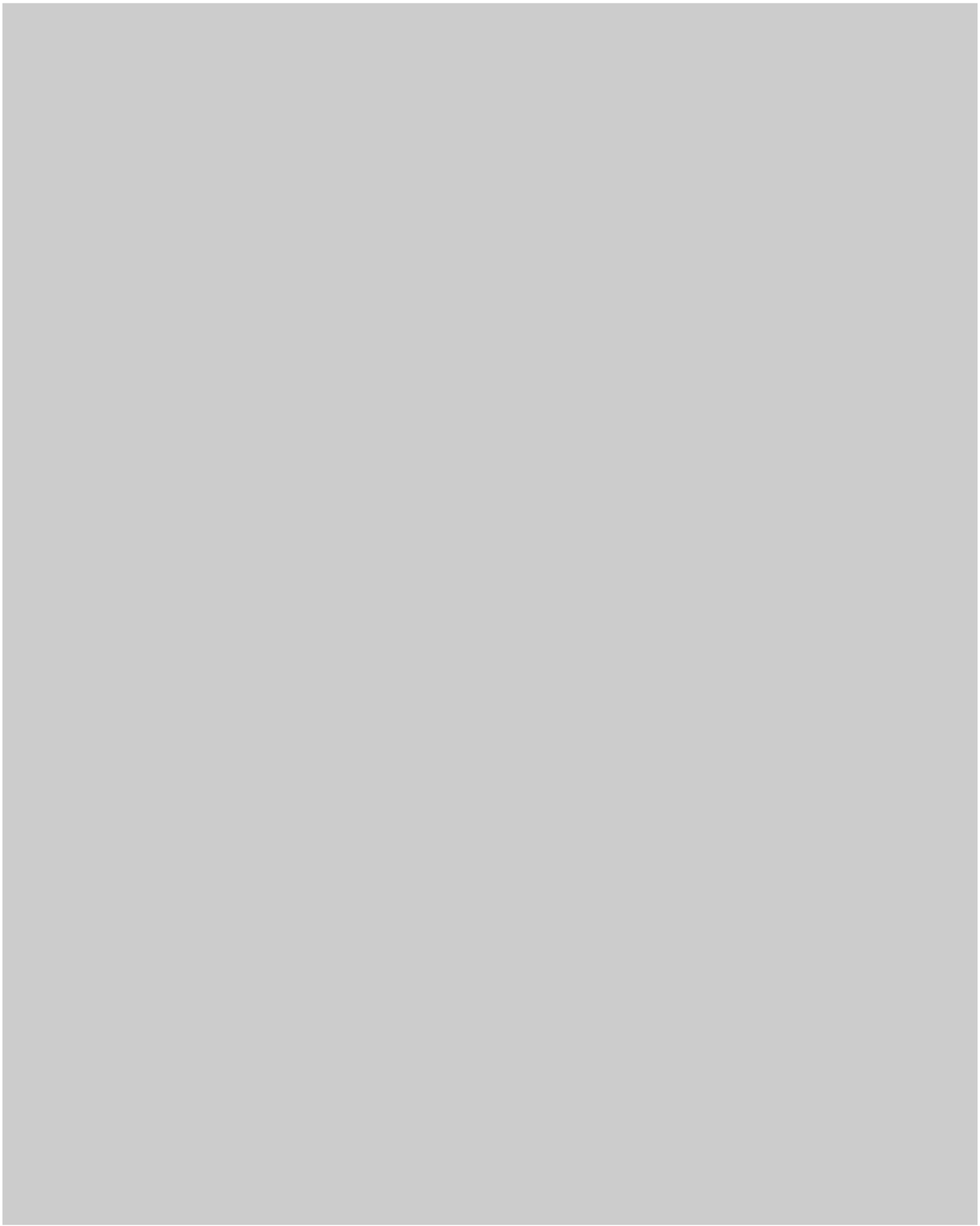
PART I

Introduction to 3D Game Programming

IN THIS PART

CHAPTER 1	Getting Started with 3D Game Programming	***9***
CHAPTER 2	A Crash Course in Windows and DirectX	***69***
CHAPTER 3	3D Game Programming with a Virtual Computer	**115**

Playrix Ex. 1023, page 32



CHAPTER 1

Getting Started with 3D Game Programming

"1.21 Gigawatts!!!"

—Dr. Emmett Brown, *Back to the Future*

In this chapter we're going to cover some general game programming topics, like game loops and the differences between 2D and 3D games, to get you warmed up. Then we'll finish off with a little 3D game to get you set up on the compiler and DirectX correctly. If you have already read the first book of my grand series, *Tricks of the Windows Game Programming Gurus*, you can probably skim this chapter and only focus on the content at the end. But if you haven't, you should definitely read it—even if you're an intermediate or advanced game programmer. Here's what's in store:

- A brief introduction
- The elements of a 2D/3D game
- General game programming guidelines
- Using tools
- A sample 16-bit/windowed game: *Raiders 3D*

A Brief Introduction

This book is really the second volume of a two- (possibly three) volume series about 2D and 3D game programming. The first volume, *Tricks of the Windows Game Programming Gurus*, primarily covered the following:

- Windows programming
- The Win32 API

IN THIS CHAPTER

- A Brief Introduction
- The Elements of a 2D/3D Game
- General Game Programming Guidelines
- Using Tools
- A Sample 3D Game: *Raiders 3D*

Playrix Ex. 1023, page 34

03 0672318350 CH01 05•02•2003 2:32 PM Page 10

- DirectX foundation
- Artificial intelligence
- Basic physics modeling
- Sound and music
- Algorithms
- Game programming
- 2D raster and vector graphics

This book is really just a continuation of the previous book. However, I've attempted to write it so that if you haven't read the first book, you can still get a lot out of it in relation to real-time 3D graphics as it pertains to writing 3D games. Hence, the ideal reader of the book is either someone that has read the first *Tricks* book and is interested in 3D, or someone that is already a 2D game programmer and wants to learn 3D techniques from a software and algorithmic point of view.

With that in mind, I'm going to focus on 3D math and graphics in this book, and move very fast over anything related to game programming. I assume that you know it—if you don't, I suggest picking up *Tricks of the Windows Game Programming Gurus* (or any other good game programming book) and sitting at the computer until you feel comfortable with Windows, DirectX, and game programming in general.

On the other hand, even if you haven't read the previous *Tricks* and know nothing of game programming, you will still get something out of this book, because this is a graphics book. We are going to build the 3D engine(s) for the book chapter by chapter, but to save time (and about 1,500 pages), we're going to start off with the basic DirectX engine developed in the first book. (This will help with 2D, sound, music, input, and so forth.) Of course, this engine is a DirectX 7–8 engine, and now DirectX has changed a bit in Version 8.0+ to make 2D harder, because DirectDraw has been merged with Direct3D. We are going to stick with the DirectX 7 and 8 interfaces, but compile under the DirectX 9.0 SDK.

NOTE

This book is about software and algorithms, not DirectX, so I could write it in DOS and the material would be applicable. We just need reasonably advanced graphics, I/O, and a sound layer, so DirectX 8.0+ is fine. In other words, if you're a fan of Linux, porting the engine to SDL will be a snap!

Therefore, if you've already read the previous *Tricks*, the engine and functions will be very familiar. If you haven't, you'll just have to think of it as a black box API (source included, of course), because in this book we'll be adding the 3D aspect to it as we go.

Playrix Ex. 1023, page 35

03 0672318350 CH01 05•02•2003 2:32 PM Page 11

Don't worry—in the next chapter, we'll cover the entire 2D engine API and structure, and you'll know what every function does; I'll also throw you a number of demos to get you warmed up using the basic DirectX engine created in the first book.

In this book, I want to try and take as much of a generic, or better yet, "virtual" point of view of the graphics system. Although the code will be based on my previous engine from the first *Tricks*, the point is that my previous engine does nothing more than set up a double-buffered graphics system in a standard, linearly addressable manner.

Thus, if you want to port any of the code to a Mac or Linux box, it should literally take anywhere from a few hours to a weekend. My goal here is to teach 3D graphics and math in general. It just so happens that the Windows platform under DirectX is the most prevalent computing system in the world at this time, so that's what I based the low-level stuff on. It's the high-level concepts that we are interested in here.

I have written about 20 or so 3D engines in my life, and I have them all here on my hard drives. But when I write a book, I like to write a new engine as I (we) go, so that I write the engine for the book, and not a book about that engine. The point of this book is not to regurgitate a 3D engine, but to make one. Thus, I'm not sure what we will end up with! In fact, I'm kind of excited to see what will happen. I'm shooting for a *Quake*-like 6DOF (degree of freedom) engine, but who knows? You will learn everything necessary to make a *Quake* engine, but I might get really focused on outdoor engines or something else—who knows. The point is that it will be a learning experience, and not the typical code dump with author annotations. I'm sure you know what I'm talking about!

Lastly, readers from the first book might notice that some of the material is the same here and there. Truthfully, I can't just jump into 3D graphics without some overlap with the last book. I can't assume someone has my other book, or force them to buy it (it's not like it makes me rich or anything, but maybe they don't have access). Whatever the case, the first few chapters will have a bit of overlap here and there in relation to DirectX, the engine, and Windows.

The Elements of a 2D/3D Game

As a review, let's take a look at what makes a video game program different from any other kind of program. Video games are extremely complex pieces of software. In fact, they are without a doubt the hardest programs to write. Sure, writing something like MS Word is harder than writing an asteroids game, but writing something like *Unreal*, *Quake Arena*, or *Halo* is harder than any program I can think of—including military weapons control software!

This means that you have to learn a new way of programming that's more conducive to real-time applications and simulation, rather than single-line, event-driven, or sequential logic programs that you might be used to. A video game is basically a continuous loop that performs logic and draws an image on the screen—usually at a rate of 30–60 fps (frames per second) or more. This is similar to how a movie is displayed, except that you are making the movie as you go.

Playrix Ex. 1023, page 36

03 0672318350 CH01 05•02•2003 2:32 PM Page 12

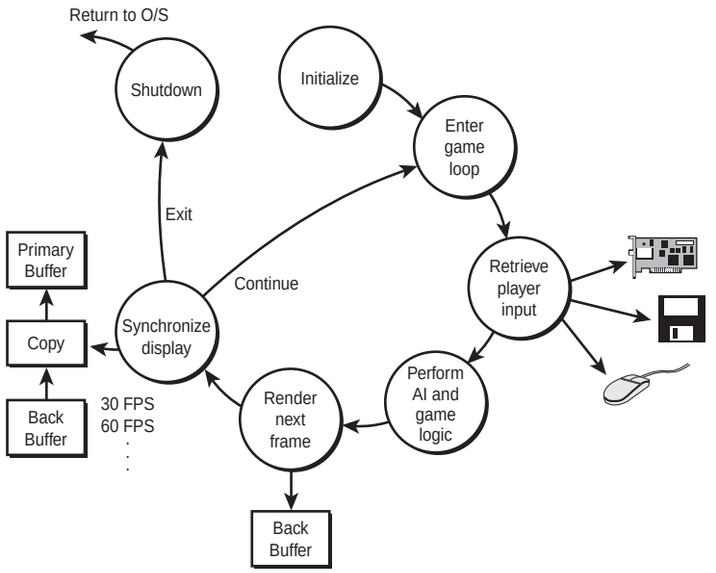


FIGURE 1.1 General game loop architecture.

Let's begin by taking a look at a simplified game loop, as shown in Figure 1.1. The following sections describe each section.

Section 1: Initialization

In this section, you perform the standard operations you would for any program, such as memory allocation, resource acquisition, loading data from the disk, and so forth.

Section 2: Enter Game Loop

In this section, the code execution enters into the main game loop. This is where the action begins and continues until the user exits out of the main loop.

Section 3: Retrieve Player Input

In this section, the player's input is processed and/or buffered for later use in the AI and logic section.

Section 4: Perform AI and Game Logic

This section contains the majority of the game code. The artificial intelligence, physics systems, and general game logic are executed, and the results are used to draw the next frame on the screen.

Playrix Ex. 1023, page 37

03 0672318350 CH01 05•02•2003 2:32 PM Page 13

Section 5: Render Next Frame

In this section, the results of the player's input and the execution of the game AI and logic are used to generate the next frame of animation for the game. This image is usually drawn on an offscreen buffer area, so you can't see it being rendered. Next the image is very quickly copied to the visible display, creating the illusion of animation. In the case of a 3D software-based engine, thousands (or millions in some cases) of polygons are rendered that compose the world by a very complex 3D graphics pipeline. In the case of a 3D hardware-accelerated engine based on OpenGL or Direct3D, much of the work is offloaded to the hardware.

Section 6: Synchronize Display

Many computers will speed up or slow down due to the level of complexity that the game is currently exhibiting. For example, if there are 1,000 objects running around on the screen, the CPU is going to have a higher load than if there are only 10 objects. Therefore, the frame rate of the game will vary, which isn't acceptable. Hence, you must synchronize the game to some maximum frame rate and try and hold it there using timing and/or wait functions. Usually, 30 fps is considered to be minimum frame rate these days, with 60 fps being ideal. Frame rates over 60 fps are hardly an improvement, because your brain has problems processing information any faster that 60 fps—this is a waste, unless you are a Borg.

NOTE

Even though some games ideally update each frame 30–60 times a second, this frame rate might drop as the complexity level of the 3D rendering increases. However, using time-based calculations in the game logic, physics, and AI sections of the code will at least attempt to keep the display somewhat steady, as far as how much objects move in real time. That is, if time (*t*) is used in each of the calculations for motion and animation control, the result is much more temporally consistent animation.

Section 7: Loop

This section is fairly simple—just loop back to the beginning of the game loop and do it all again.

Section 8: Shutdown

This is the end of the game, meaning that the user has exited the main body or game loop and wants to return to the operating system. However, before doing this, you must release all resources and clean up the system, as you would for any other piece of software.

Now you might be wondering about all the details of a real game loop. Granted, the preceding explanation is a little oversimplified, but it captures the essence of what's going on. In reality, the game loop will in most cases be a *FSM (finite state machine)* that contains

Playrix Ex. 1023, page 38

03 0672318350 CH01 05•02•2003 2:32 PM Page 14

a number of states. For example, the following listing is a more detailed version of what a C/C++ game loop might look like in real code.

LISTING 1.1 A Simple Game Event Loop

```
// defines for game loop states
#define GAME_INIT    // the game is initializing
#define GAME_MENU    // the game is in the menu mode
#define GAME_START   // the game is about to run
#define GAME_RUN    // the game is now running
#define GAME_RESTART  // the game is going to restart
#define GAME_EXIT    // the game is exiting

// game globals
int game_state = GAME_INIT; // start off in this state
int error   = 0;   // used to send errors back to OS


// main begins here

void main()
{
// implementation of main game-loop

while (game_state!=GAME_EXIT)
  {
  // what state is game loop in
    switch(game_state)
  {
    case GAME_INIT: // the game is initializing
       {
       // allocate all memory and resources
      Init();

      // move to menu state
      game_state = GAME_MENU;
      } break;

     case GAME_MENU: // the game is in the menu mode
       {
       // call the main menu function and let it switch states
       game_state = Menu();
```

Playrix Ex. 1023, page 39

03 0672318350 CH01 05•02•2003 2:32 PM Page 15

LISTING 1.1 Continued

```
    // note: we could force a RUN state here
    } break;

  case GAME_START:  // the game is about to run
    {
    // this state is optional, but usually used to
    // set things up right before the game is run
    // you might do a little more housekeeping here
    Setup_For_Run();

    // switch to run state
    game_state = GAME_RUN;
    } break;

  case GAME_RUN:  // the game is now running
     {
     // this section contains the entire game logic loop

     // clear the display
     Clear();

     // get the input
     Get_Input();

     // perform logic and ai
     Do_Logic();

     // display the next frame of animation
     // render 2D or 3D world
     Render_Frame();

     // synchronize the display to 30 fps+
     Wait();

     // the only way that state can be changed is
     // thru user interaction in the
     // input section or by maybe losing the game..
     } break;

     case GAME_RESTART: // the game is restarting
       {
       // this section is a cleanup state used to
```

Playrix Ex. 1023, page 40

03 0672318350 CH01 05•02•2003 2:32 PM Page 16

LISTING 1.1 Continued

```
          // fix up any loose ends before
          // running again
          Fixup();
          // switch states back to the menu
          game_state = GAME_MENU;
          } break;

       case GAME_EXIT:  // the game is exiting
          {
          // if the game is in this state then
          // it's time to bail, kill everything
          // and cross your fingers
          Release_And_Cleanup();

          // set the error word to whatever
          error = 0;

          // note: we don't have to switch states
          // since we are already in this state
          // on the next loop iteration the code
          // will fall out of the main while and
          // exit back to the OS
          } break;

    default: break;
    } // end switch

  } // end while

// return error code to operating system
return(error);

} // end main
```

Although the code is nonfunctional, I think that you can get a good idea of the structure of a real game loop by studying it. All game loops—2D, 3D, whatever—pretty much follow this structure in one way or another. Take a look at Figure 1.2 to see the state transition diagram for the game loop logic. As you can see, the state transitions are fairly sequential.

Playrix Ex. 1023, page 41

03 0672318350 CH01 05•02•2003 2:32 PM Page 17

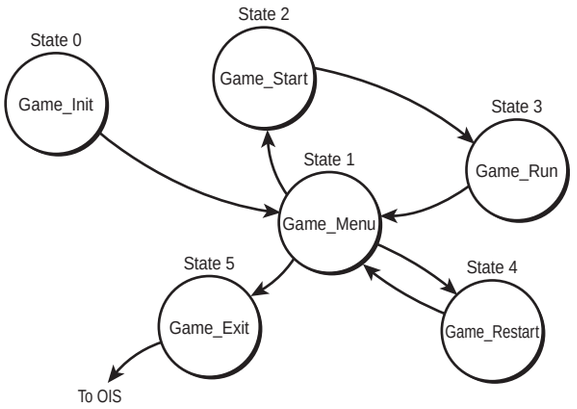


FIGURE 1.2 State transition diagram for the game loop logic.

General Game Programming Guidelines

The next thing I want to talk about is general game programming techniques and philosophies that you should think about and try to adopt (if you can) that will make game programming much easier for you.

To begin with, video games are ultra-high performance computer programs. This means that no longer can you use high-level APIs for time-critical or memory-critical code sections. You must, for the most part, write everything that is related to the inner loop of your game code yourself, or your game will suffer terrible speed and performance problems. Obviously, this doesn't mean that you can't trust APIs like DirectX, because DirectX was written to be as high performance and "thin" as possible. But in general, avoid high-level function calls based on the Win32 API. For example, you might think that `memset()` is very fast, but it only fills using single `BYTE`s. A much better fill would use `QUAD`s, so that four `BYTE`s can be written at once. For example, here's an inline assembly language function that I use to do `QUAD-BYTE` fills:

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
// this function fills or sets unsigned 32-bit aligned memory
// count is number of quads

_asm
  {
```

Playrix Ex. 1023, page 42

03 0672318350 CH01 05•02•2003 2:32 PM Page 18


```
  mov edi, dest  ; edi points to destination memory
  mov ecx, count ; number of 32-bit words to move
  mov eax, data  ; 32-bit data
  rep stosd    ; move data
  } // end asm

} // end Mem_Set_QUAD
```

And here's a WORD, or 2-BYTE version:

```
inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
// this function fills or sets unsigned 16-bit aligned memory
// count is number of words

_asm
  {
  mov edi, dest  ; edi points to destination memory
  mov ecx, count ; number of 16-bit words to move
  mov ax, data  ; 16-bit data
  rep stosw    ; move data
  } // end asm

} // end Mem_Set_WORD
```

These few lines of code can speed the frame rate of your game by two to four times in some cases! So, you definitely need to know what's inside of an API call if you are going to use API calls.

NOTE

Additionally, the Pentium III, 4, and greater support SIMD (Single Instruction Multiple Data) instructions that allow parallel processing of simple mathematical operations, so there's a lot of room to optimize basic math operations, like vector mathematics and matrix multiplications, but we'll get to that later.

Let's take a look at a list of tricks to keep in mind as you're programming.

TRICK

Back up your work all the time. When writing game code, you are going to lock the system up fairly frequently. Redoing a sorting algorithm is one thing, but redoing the AI for a character and the collision detection is another.

Playrix Ex. 1023, page 43

03 0672318350 CH01 05•02•2003 2:32 PM Page 19

TRICK

When you start on your game projects, be organized. Use reasonable filenames and directory names, come up with a consistent variable naming convention, and try to use separate directories for graphics and sound data rather than dumping everything in one directory.

TRICK

Use inline functions. You can get rid of the function call completely by using the inline directive. The inline directive instructs the compiler to make its best attempt to put the code for the function right where it's called, rather than making the actual function call. Of course, this makes bigger programs, but speed is more important. Here's an example:

```
inline void Plot_I(int x, int y, int color)
{
// plots a pixel on the screen
video_buffer[x + y*MEMORY_PITCH] = color;
} // end Plot_I
```

Notice in this case I didn't use globals, because the compiler will in effect perform the same type of data aliasing. However, globals would come in handy if only one or two of the parameters were changing between calls, because the old values can be used without reloading.

TRICK

Always use 32-bit variables rather than 8- or 16-bit when possible. The Pentium and later processors are totally 32-bit, and the Itanium is 64-bit. This means that they don't like 8- or 16-bit data words. In fact, smaller data can slow them down because of caching and other related memory addressing anomalies. For example, you might create a structure that looks something like this:

```
struct CPOINT
{
short x,y;
unsigned char c;
} // end CPOINT
```

Although this might seem like a good idea, it's not! First, the structure itself is now 5 bytes long *(2*sizeof(short) + sizeof(char))* = 5. This is really bad, and will wreak havoc on the memory addressing. A better approach is the following structure:

```
struct CPOINT
{
int x,y;
int c;
} // end CPOINT
```

Playrix Ex. 1023, page 44

03 0672318350 CH01 05•02•2003 2:32 PM Page 20

NOTE

STRUCTs in C++ are just like CLASSes, except that they have default PUBLIC visibility.

This new structure is much better. For one thing, all the elements are the same size—that is, `sizeof(int)` = 4 bytes. Therefore, a single pointer can be incremented on a DWORD boundary to access any member. Of course, the new structure is now `(3*sizeof(int))` = 12 bytes, but at least it's a multiple of 4 or on a DWORD boundary; this is definitely going to improve performance.

In fact, if you really want to make things rock, you can pad all structures to make them a multiple of 32 bytes. This is the optimal length because of standard on-chip cache line sizes in the Pentium class processors. Padding can be accomplished manually by adding dummy variables, or with a compiler directive (the easy way). Of course, this might waste a lot of memory, but it might be worth the speed.

TRICK

Don't be afraid to use global variables. Many video games don't use parameters for a lot of time-critical functions, and simply use a global parameter-passing area instead. For example, if a function looks like this:

```
void Plot(int x, int y, int color)
{
// plots a pixel on the screen
video_buffer[x + y*MEMORY_PITCH] = color;
} // end Plot
```

The body of the function takes less time than the function call! This is because of the parameter pushing and popping on the stack. In this case, a better method might be to first create a global parameter-passing area, and then make assignments before a call:

```
int gx,gy,gz,gcolor; // define some globals

void Plot_G(void)
{
// plot a pixel using globals
video_buffer[gx + gy*MEMORY_PITCH] = gcolor;

} // end Plot_G
```

TRICK

Program in a RISC (Reduced Instruction Set Computer)-like manner. Make your code simple rather than complex. Pentium processors especially like simple instructions rather than complex ones. Also, making your code longer with simpler instructions makes it easier for the compiler. For example, don't do this:

Playrix Ex. 1023, page 45

03 0672318350 CH01 05•02•2003 2:32 PM Page 21


```
if (x+=(2*buffer[index++])>10)
{
// do work
} // end if
```

Do this:

```
x+=(2*buffer[index]);
index++;

if (x > 10)
{
// do work
} // end if
```

There are two reasons for coding like this. First, it allows a debugger to put break points in between code sections, and second, it makes it easier for the compiler to send simplified code to the PentiumX, which allows it to process more code in parallel using multiple execution units. Complex code is bad!

TRICK

Use binary shifts for the simple multiplication of integers by powers of two. Because all data in a computer is stored in binary form, shifting the bit pattern to the left or right is equivalent to multiplication and division, respectively. For example:

```
int y_pos = 10;

// multiply y_pos by 64
y_pos = (y_pos << 6); // 2^6 = 64
```

Similarly,

```
// to divide y_pos by 8
y_pos = (y_pos >> 3); // 1/2^3 = 1/8
```

Of course, these days (especially on the Pentium 4), there are single-cycle multiplications, but shifting is always a safe bet to multiply and divide by two really fast.

TRICK

Write efficient algorithms. All the assembly language in the world isn't going to make an $O(n^2)$ algorithm go faster. It's better to use clean, efficient algorithms rather than brute force.

Playrix Ex. 1023, page 46

03 0672318350 CH01 05•02•2003 2:32 PM Page 22

TRICK

Don't optimize your code as you program. When writing game code, it's tempting to optimize as you go. This is usually a waste of time. Wait until you are done with a major code block or until you're done with the game before you start heavy optimization (Intel's VTUNE is a great tool for profiling). This will save you time in the end, because you won't have to deal with cryptic code or optimizations that aren't necessary. When the game is done, that's when you start profiling and finding problem areas to optimize. On the other hand, don't program sloppily.

TRICK

Use C++ sparingly. If you're a seasoned professional, go ahead and do as you please, but don't go class crazy or overload everything to death. In the end, simple, straightforward code is the best and easiest to debug. And I never want to see multiple inheritance! If C is good enough for *Doom* and *Quake*, there's no Earth-shattering reason to use C++. If it helps you, great; if not, stay away from it for game programming until you are comfortable with game programming and C++.

TRICK

If you see that you're going down a rocky road, stop, back up, and take a detour. I have seen many game programmers start down a bad programming line and bury themselves. It's better to realize you've made a mistake and redo 500 lines of code than to have a generally undesirable code structure. So, if you see a problem with what you're doing, re-evaluate and make sure that the time you're saving is worth it.

TRICK

Don't write a lot of complex data structures for simple objects. Just because linked lists are cool doesn't mean you should use them for a fixed array that you know will always be around 256 items. Just allocate it statically and be done with it. Video game programming is 90% data manipulation. Keep your data as simple and visible as possible, so you can access it quickly, do what you need to, and move on. Make sure the data structure fits the problem.

Using Tools

Writing video games used to be nothing more than a text editor and maybe homemade paint and sound programs. However, today things are a little more complicated. To write a 3D game, at a minimum you need a C/C++ compiler, 2D paint program, sound processing program, and some kind of minimal 3D modeler (unless you're going to enter all the 3D model data as ASCII data). Additionally, you'll need a music sequencing program if you're going to use any MIDI, or at least make changes to presequenced MIDI.

Playrix Ex. 1023, page 47

03 0672318350 CH01 05•02•2003 2:32 PM Page 23

Let's take a look at some of the more popular products and what they do.

- C/C++ compilers—For Windows 9*x*/Me/XP/2000/NT development, there simply is no better compiler than MS VC++, shown in Figure 1.3. It does everything you need it to do and more. The .EXEs generated are the fastest code available. The Borland compiler will also work fine (and is a bit cheaper), but has a much smaller feature set. In either case, you don't need the full-blown version of either—a student version that makes Win32 .EXEs is more than enough.

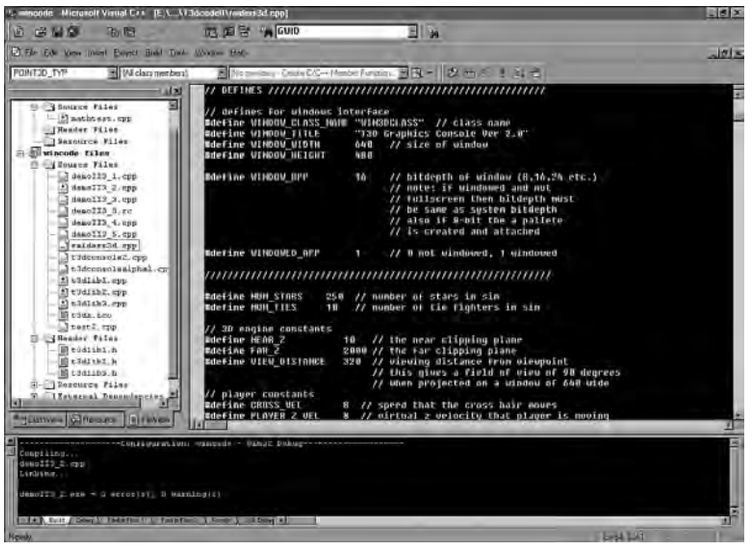


FIGURE 1.3 The Microsoft VC++ environment.

- 2D bitmap art software—Here you have paint programs, drawing programs, and image processing. Paint programs primarily allow you to draw images pixel by pixel with primitives and to manipulate them. As far as I'm concerned, Paint Shop Pro by JASC, shown in Figure 1.4, is the leader of the pack for price versus performance. But the ultimate favorite of most artists is Adobe Photoshop—it has more firepower than you'll ever need.
- 2D vector art software—Drawing programs, on the other hand, allow you to create images that are mostly constructed from curves, lines, and 2D geometrical primitives. The types of programs aren't as useful, but if you need one, Adobe Illustrator is the way to go.

Playrix Ex. 1023, page 48

03 0672318350 CH01 05•02•2003 2:32 PM Page 24

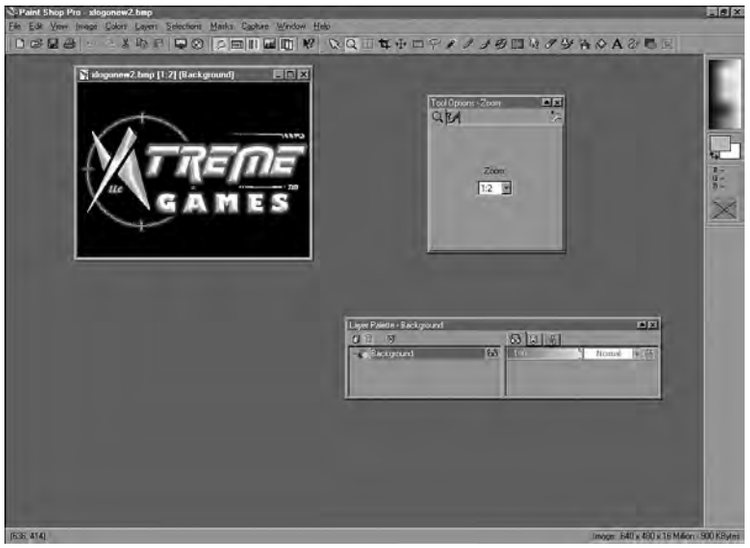


FIGURE 1.4 Paint Shop Pro by JASC Software.

- Post-production image processing—Image processors are the final class of 2D art programs. These programs are more for post-production rather than art creation. In this area, Adobe Photoshop is the favorite in most circles, but I think Corel Photo-Paint is better—you decide.
- Sound processing software—Ninety percent of all sound FX used in games today are digitized samples. To work with sound data of this type, you're going to need a digital sound processing program. One of the best programs in this genre is Sound Forge, shown in Figure 1.5. It is by far one of the most complex sound processing programs I have ever seen, and ironically the simplest to use as well. I'm still trying to figure out all the cool features! Cool Edit Pro is also very powerful, but I have less experience with it.
- 3D modelers—This is where things get financially challenging. 3D modelers can cost in the tens of thousands of dollars, but recently a number of low-cost modelers have shown up that have enough power to literally make a movie. The modeler that I primarily use for simple to medium-scale 3D models and animation is Caligari trueSpace, shown in Figure 1.6. This is the best 3D modeler for the price—it only costs a few hundred dollars, and has the best interface there is, in my opinion.

If you want a little more firepower and absolute photo-realism, 3D Studio Max, shown in Figure 1.7, is the way to go, but it costs around $2,500 or more, so that might be something to think about. However, because we are going to use these modelers just to create 3D meshes for the most part, and not for rendering, all the bells and whistles aren't really needed—trueSpace is the way to go, or perhaps a freeware or shareware modeler.

Playrix Ex. 1023, page 49

03 0672318350 CH01 05•02•2003 2:32 PM Page 25

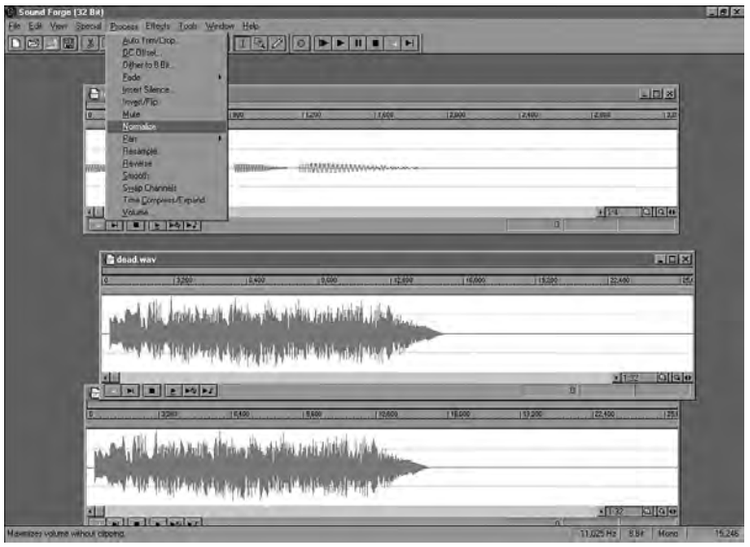


FIGURE 1.5 Sound Forge in action.

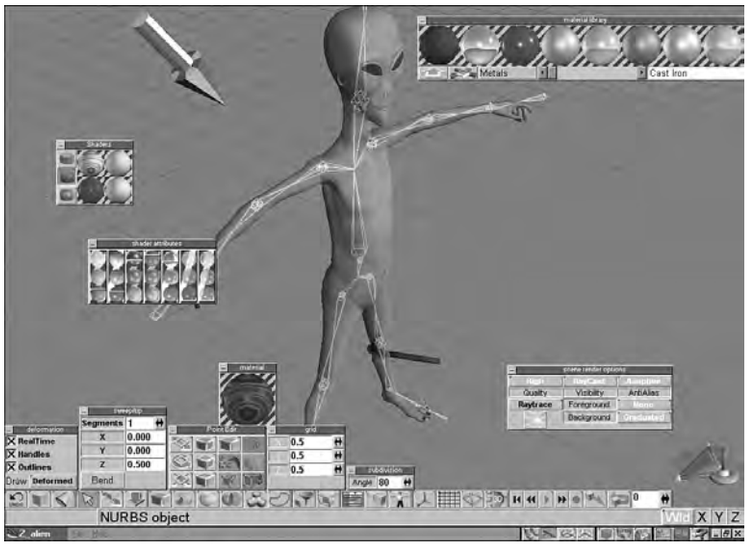


FIGURE 1.6 Caligari's trueSpace 3D modeler.

Playrix Ex. 1023, page 50

03 0672318350 CH01 05•02•2003 2:32 PM Page 26

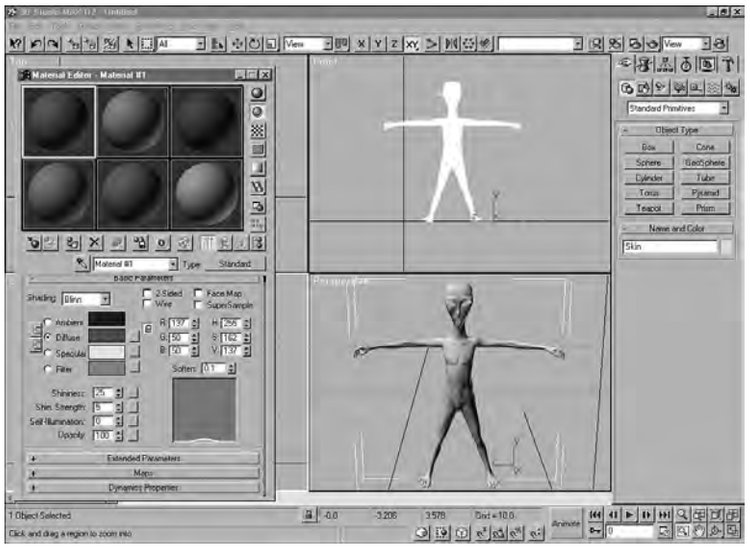


FIGURE 1.7 3D Studio Max (Heavy Metal).

NOTE

The Internet is full of free or shareware 3D modelers that are in some cases as powerful as the real thing. So, if cash is a problem, hunt around and see what you can find.

3D Level Editors

In this book, we are going to write a software-based 3D engine. However, we aren't going to write a complex tool to create 3D interior models. It's possible to create a 3D world with a 3D modeler, but there are much better programs to do this, such as WorldCraft, shown in Figure 1.8. Hence, the best approach to use when writing a 3D game and engine is to use a file format and data structure that is compatible with the types output by a commonly available editor such as WorldCraft (or a similar editor), so you can use someone else's tool to create your worlds. Of course, the file format of most of these editors is ultimately based on the work of id Software and the *Quake* engine, but whether it's based on the *Quake* file format or whatever, it's a proven file format, and obviously works well.

Playrix Ex. 1023, page 51

03 0672318350 CH01 05•02•2003 2:32 PM Page 27

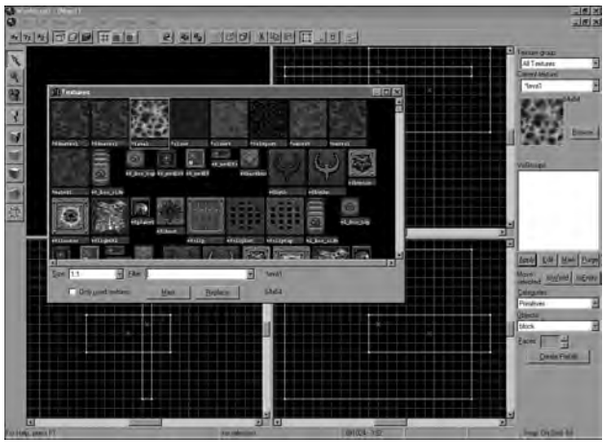


FIGURE 1.8 The WorldCraft level editor.

- Music and MIDI sequencing programs—There are two kinds of music in today's games: pure digital (like a CD), and MIDI (Musical Instruments Digital Interface), which is a synthesized performance based on note data. If you want to manipulate MIDI information and songs, you'll need a sequencing package. One of the best is called CakeWalk, shown in Figure 1.9. CakeWalk is also reasonably priced, so I suggest that you look into this program if you plan on recording and manipulating any MIDI music. We'll talk about MIDI data when covering DirectMusic.

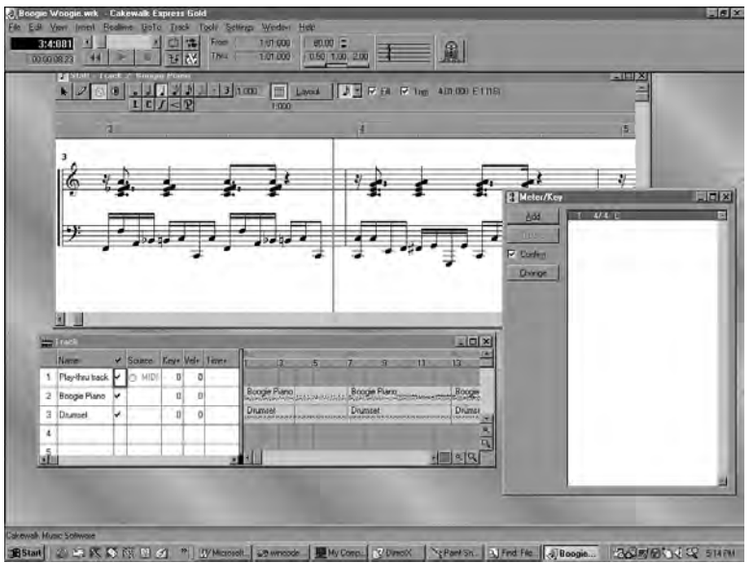


FIGURE 1.9 CakeWalk MIDI track sequencer.

Playrix Ex. 1023, page 52

03 0672318350 CH01 05•02•2003 2:32 PM Page 28

NOTE

And now for the cool part … A number of the manufacturers of the software listed previously have allowed me to put shareware or evaluation versions on the CD, so make sure to check them out!

Setting Up to Get Down—Using The Compiler

One of the most frustrating parts of learning 3D windows game programming is learning how to use the compiler. In most cases, you're so excited to get started that you dive into the IDE and try to compile, and a million compiler and linker errors pop up! To help with this problem, let's cover a few basic compiler concepts here.

NOTE

The details listed here are Visual C++-specific, but the ideas are the same for any compiler.

1. Read the entire compiler manual—please, please, I beg you!
2. You must install the DirectX 9.0 SDK on your system. To do this, navigate into the DirectX SDK directory on the CD, read the `README.TXT`, and do what it says, which should be nothing more than click on the DirectX SDK installation programs. And remember, we are using DirectX 9.0 for this book (but only using the interfaces from 7 and 8), so you need to use the DirectX SDK on the CD. However, you can compile everything with DirectX 8.0, if you really want to.
3. We are going to make Win32 .EXE application programs, ***not*** .DLLs, ActiveX components, CONSOLE applications, or anything else (unless I tell you to). So the first thing you need to do with your compiler if you want to compile is create a new project or workspace and set the target output file to Win32 .EXE. This step is shown for the Visual C++ 6.0 compiler in Figure 1.10.

NOTE

There are two types of .EXEs that you can build when compiling: Release and Debug. Release is faster code and optimized, whereas Debug is slower and has debugger hooks in it. I suggest that while you are developing, you build the Debug version until it works, then go ahead and set the compiler to Release mode.

4. Add the source files to the project using the Add Files command from the main menu, or from the project node itself. This is shown for the Visual C++ 6.0 compiler in Figure 1.11.

Playrix Ex. 1023, page 53

03 0672318350 CH01 05•02•2003 2:32 PM Page 29

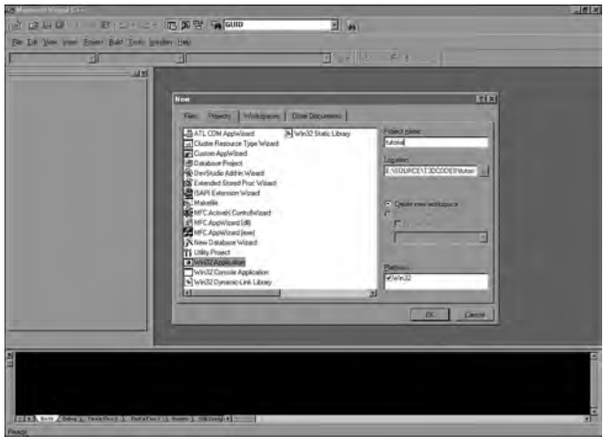


FIGURE 1.10 Creating a Win32 .EXE with Visual C++ 6.0.

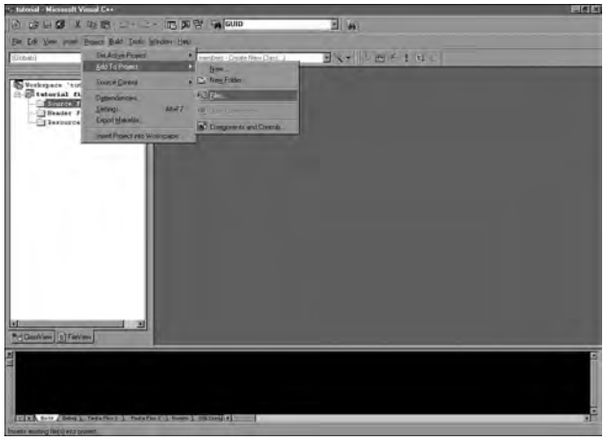


FIGURE 1.11 Adding files to a project with Visual C++ 6.0.

5. When you compile anything that needs DirectX, you're going to have to include in the search path of the compiler the location of the DirectX header files, as well as the location of the DirectX .LIB files. You can make these additions under the Options, Directories menu item in the main menu of the IDE. Then make the appropriate additions to the Include Files and Library Files variables. This is shown in Figure 1.12. (Of course, make sure you point them to your particular installation of DirectX and the newest libraries and header files.)

Playrix Ex. 1023, page 54

03 0672318350 CH01 05•02•2003 2:33 PM Page 30

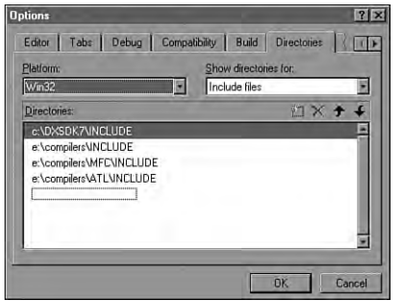


FIGURE 1.12 Setting up the search paths in Visual C++ 6.0.

NOTE

Make sure that the DirectX search nodes are **FIRST** in the list(s). You don't want the compiler finding old versions of DirectX that might have come with the compiler.

6. Additionally, you have to actually include the DirectX COM interface import libraries, listed here and shown in Figure 1.13, to the link list of your project or to the project resource list.

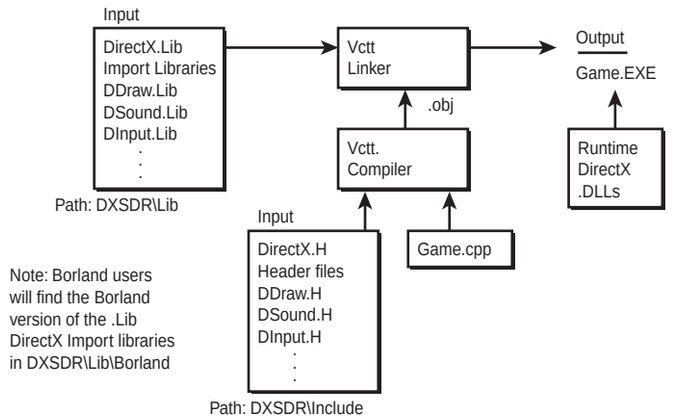


FIGURE 1.13 The resources needed to create a Win32 DirectX application.

- `DDRAW.LIB`
- `DSOUND.LIB`
- `DINPUT.LIB`

Playrix Ex. 1023, page 55

03 0672318350 CH01 05•02•2003 2:33 PM Page 31

- DINPUT8.LIB
- Any others that I mention for a particular example

These DirectX .LIB files are located in the LIB\ directory wherever you installed the DirectX SDK. **You MUST add these .LIB files to your project or workspace**. You can't just add the LIB\ directory to the search path alone—that simply indicates where to search, but doesn't tell the compiler/linker to actually link the DirectX .LIB files. I have received thousands (that's right, thousands) of emails from people that don't do this and have problems. Once again: **You must manually include the DirectX .LIB files in the compiler's link list with the other libraries OR in the project along with your .CPP files**. You can do this in the Project, Settings submenu in the Link, General dialog in the Object/Library-Modules list. This is shown in Figure 1.14.

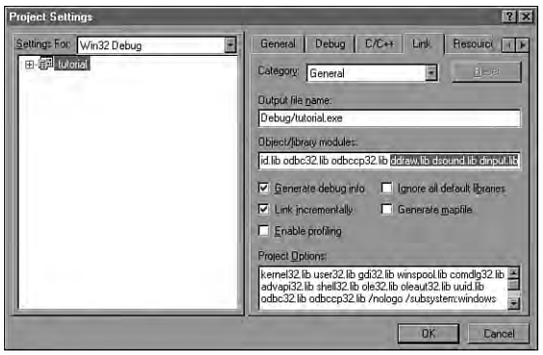


FIGURE 1.14 Adding the DirectX .LIB files to the link list in Visual C++ 6.0.

NOTE

While you're at it, you might have to add the Windows Multimedia Extensions library WINMM.LIB to your project if you're using Visual C++. This file is located in the LIB\ directory of your Visual C++ compiler installation. If it's not there, just use the FIND command from the Start menu to find it. Once found, add it to the link list.

7. You're ready to compile your programs.

NOTE

If you're a Borland user, there is a separate BORLAND\ library directory within the DirectX SDK, so make sure to add the .LIB files found in there for DirectX and NOT the higher Visual C++ DirectX .LIB files up higher in the directory tree, which are MS-Visual C++ compatible.

Playrix Ex. 1023, page 56

03 0672318350 CH01 05•02•2003 2:33 PM Page 32

If you still have questions about this, don't worry—I will revisit the steps a number of times throughout the book when compiling the programs. But if I get one email about compiling and it's something that is right here in the preceding paragraphs—I'm going to explode!

> **NOTE**
>
> I'm sure you have heard of the new Visual .NET System. Technically, it's the newest of Microsoft's compiler and development technologies, but it's brand new. It only runs on Windows XP/2000 decently, and is overkill for our purposes, so we will stick with Visual C++ 6.0. However, everything works with .NET just as well: Win32 is Win32!

A Sample 3D Game: *Raiders 3D*

Before we both lose our minds talking about math, 3D game programming, and graphics, I would like to take a pause and show you a complete 3D space game—a simple one, but a game nonetheless. This way you can see a real game loop, some graphics calls, and give a shot at compilation. Sound good, my young Jedi?

The problem is we are only on Chapter 1, so it's not like I can use stuff from later chapters—that would be cheating, right? And I haven't showed you the API engine from the previous *Tricks*. So, I've decided to get you accustomed to using "black box" APIs for game programming. Hence, this sample game will get you accustomed to the idea of using an API such as DirectX and the engine from the first book.

> **NOTE**
>
> Although in the 70s, 80s, and early 90s you could get away with doing everything yourself, in the 21st century, there are simply too many hardware and software subsystems in a computer—it's nearly impossible to write everything yourself. Alas, we finally have to give in as game programmers and use a few APIs such as Win32, DirectX, and so forth.

Therefore, based on the "black box" requirement, I wondered what, at the absolute minimum, would be needed to create a 16-bit 3D wireframe space game?

All we really need is the following functionality from an API:

- Change into any graphics mode using DirectX
- Draw colored lines and pixels on the screen
- Get the keyboard input
- Play a sound from a .WAV file on disk
- Play a MIDI file from a .MID file on disk

Playrix Ex. 1023, page 57

03 0672318350 CH01 05•02•2003 2:33 PM Page 33

- Synchronize the game loop using some timing functions
- Draw strings of colored text on the screen
- Copy the double buffer, or offscreen rendering page to the display

All this functionality is of course in the game library modules `T3DLIB*`, created in the previous *Tricks*, which is composed of six files: `T3DLIB1.CPP¦H`, `T3DLIB2.CPP¦H`, and `T3DLIB3.CPP¦H`. Module 1 contains most of the DirectX stuff, Module 2 contains most of the DirectInput and DirectSound, and Module 3 contains DirectMusic for the most part.

Based on using the absolute minimum set of functions from the `T3DLIB*` game library, I wrote a game called *Raiders 3D* that demonstrates a number of the concepts that we have discussed in the chapter. Additionally, since the game is wireframe 3D, there's a lot of code that might not make any sense to you—that's good! Although I will briefly explain the 3D math and algorithms, don't take too much time trying to understand what's going on; this is just to give you some perspective.

Raiders 3D illustrates all the major components of a real 3D game, including a game loop, scoring, AI (the intelligence of slugs), collision, sound, and music. Figure 1.15 is a screenshot of the game in action. Granted, it's not *Star Wars*, but it's not bad for a few hours of work!

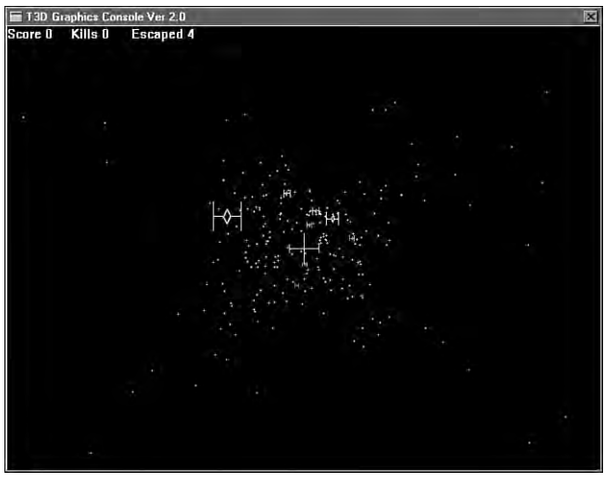


FIGURE 1.15 A screenshot of *Raiders 3D*.

NOTE

You might notice a bit of a similarity to the ancient game *Tail Gunner*. Mike, if you're reading this, did you ever get that *Tail Gunner* in your living room working?

Playrix Ex. 1023, page 58

03 0672318350 CH01 05•02•2003 2:33 PM Page 34

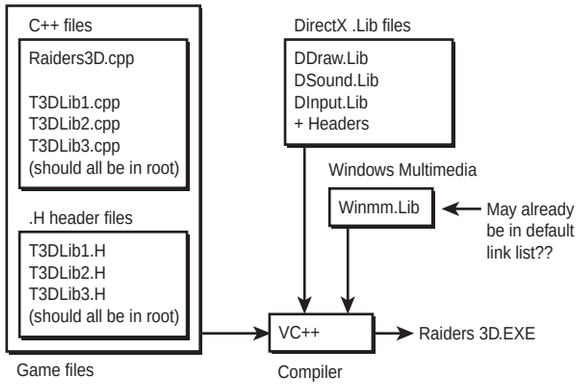


FIGURE 1.16 The code structure of *Raiders 3D*.

Before I show you the source code for the game, I want you to take a look at how the project and its various components fit together—refer to Figure 1.16. As you see from the figure, the game is composed of the following files that must be part of the project:

- `RAIDERS3D.CPP`—The main game logic that uses the functionality of `T3DLIB` and creates a minimum Win32 application.
- `T3DLIB1.CPP`—The `T3DLIB` game library source files.
- `T3DLIB2.CPP`
- `T3DLIB3.CPP`
- `T3DLIB1.H`—The header files for the game library.
- `T3DLIB2.H`
- `T3DLIB3.H`
- `DDRAW.LIB`—DirectDraw is the 2D graphics component of DirectX. The `DDRAW.LIB` import library is needed to build the application. This doesn't contain the real DirectX code; it's more of an intermediary library that you make calls to, which in turn loads the `DDRAW.DLL` dynamic link library that does the real work. You can find this in the DirectX SDK installation under `LIB\`.
- `DINPUT.LIB/DINPUT8.LIB`—DirectInput is the device input component of DirectX. The `DINPUT.LIB` and `DINPUT8.LIB` import libraries are needed to build the application. This doesn't contain the real DirectX code; it's more of an intermediary library that you make calls to, which in turn loads the `DINPUT.DLL/DINPUT8.DLL` dynamic link libraries that do the real work. You can find this in the DirectX SDK installation under `LIB\`.

Playrix Ex. 1023, page 59

03 0672318350 CH01 05•02•2003 2:33 PM Page 35

- DSOUND.LIB—DirectSound is the digital sound component of DirectX. The DSOUND.LIB import library is needed to build the application. This doesn't contain the real DirectX code; it's more of an intermediary library that you make calls to, which in turn loads the DSOUND.DLL dynamic link library that does the real work. You can find this in the DirectX SDK installation under LIB\.

NOTE

Note that there isn't a DMUSIC.LIB file, even though DirectMusic is used by T3DLIB. That's because DirectMusic is a pure COM (Component Object Model) object. In other words, there isn't an import library full of nice wrapper functions to make calls to DirectMusic—you have to do all the work yourself. Luckily I did it for you!

These files aren't needed by the compiler or linker, but are the actual DirectX runtime DLLs that will be loaded when the game application starts up:

- DINPUT.DLL/DINPUT8.DLL—These are the runtime DirectDraw libraries that actually contain the COM implementation of the DirectInput interface functions that are called through the DINPUT.LIB import library. You don't need to worry about this per se, you just need to make sure that the DirectX runtime files are installed.
- DDRAW.DLL—This is the runtime DirectDraw library that actually contains the COM implementation of the DirectDraw interface functions that are called through the DDRAW.LIB import library. You don't need to worry about this per se, you just need to make sure that the DirectX runtime files are installed.
- DSOUND.DLL—This is the runtime DirectSound library that actually contains the COM implementation of the DirectSound interface functions that are called through the DSOUND.LIB import library. You don't need to worry about this per se, you just need to make sure that the DirectX runtime files are installed.
- DMUSIC.DLL—This is the runtime DirectMusic library that actually contains the COM implementation of the DirectMusic interface functions that are called directly through COM calls. You don't need to worry about this per se, you just need to make sure that the DirectX runtime files are installed.

Based on making a few calls to the library, I have created the game RAIDERS3D.CPP, which is shown in the listing below. The game runs in windowed mode with 16-bit graphics, so make sure that your desktop is in 16-bit color mode.

Anyway, take a good look at the code: especially the main game loop, 3D math, and the calls to the game processing functions:

```
// Raiders3D - RAIDERS3D.CPP -- our first 3D game
// READ THIS!
// To compile make sure to include DDRAW.LIB, DSOUND.LIB,
```

Playrix Ex. 1023, page 60

03 0672318350 CH01 05•02•2003 2:33 PM Page 36


```
// DINPUT.LIB, WINMM.LIB in the project link list, and of course
// the C++ source modules T3DLIB1.CPP,T3DLIB2.CPP, and T3DLIB3.CPP
// in the project!!! And the T3DLIB1.H,T3DLIB2.H, and T3DLIB3.H
// header files in the working directory, so the compiler
// can find them

// to run the game make sure that your desktop is in 16bit
// color mode with a resolution of 640x480 or higher

// INCLUDES ///////////////////////////////////////////////

#define INITGUID    // make sure al the COM interfaces are available
            // instead of this you can include the .LIB file
            // DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h>  // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h> // directX includes
#include <dsound.h>
#include <dmksctrl.h>
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h" // game library includes
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// DEFINES ////////////////////////////////////////////////
```

Playrix Ex. 1023, page 61

03 0672318350 CH01 05•02•2003 2:33 PM Page 37


```
// defines for windows interface
#define WINDOW_CLASS_NAME "WIN3DCLASS" // class name
#define WINDOW_TITLE   "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH   640  // size of window
#define WINDOW_HEIGHT   480

#define WINDOW_BPP    16  // bitdepth of window (8,16,24 etc.)
                // note: if windowed and not
                // fullscreen then bitdepth must
                // be same as system bitdepth
                // also if 8-bit the a pallete
                // is created and attached

#define WINDOWED_APP   1   // 0 not windowed, 1 windowed

//////////////////////////////////////////////////////////

#define NUM_STARS  250 // number of stars in sim
#define NUM_TIES   10  // number of tie fighters in sim

// 3D engine constants
#define NEAR_Z     10  // the near clipping plane
#define FAR_Z      2000 // the far clipping plane
#define VIEW_DISTANCE  320 // viewing distance from viewpoint
               // this gives a field of view of 90 degrees
               // when projected on a window of 640 wide
// player constants
#define CROSS_VEL    8 // speed that the cross hair moves
#define PLAYER_Z_VEL  8 // virtual z velocity that player is moving
              // to simulate motion without moving

// tie fighter model constants
#define NUM_TIE_VERTS   10
#define NUM_TIE_EDGES   8

// explosiond
#define NUM_EXPLOSIONS  (NUM_TIES) // total number of explosions

// game states
#define GAME_RUNNING   1
#define GAME_OVER     0

// TYPES ////////////////////////////////////////////////
```

Playrix Ex. 1023, page 62

03 0672318350 CH01 05•02•2003 2:33 PM Page 38


```
// this a 3D point
typedef struct POINT3D_TYP
    {
    USHORT color;   // color of point 16-bit
    float x,y,z;   // coordinates of point in 3D
    } POINT3D, *POINT3D_PTR;

// this is a 3D line, nothing more than two indices into a vertex list
typedef struct LINE3D_TYP
    {
    USHORT color; // color of line 16-bit
    int v1,v2;   // indices to endpoints of line in vertex list

    } LINE3D, *LINE3D_PTR;

// a tie fighter
typedef struct TIE_TYP
    {
    int state;   // state of the tie, 0=dead, 1=alive
    float x, y, z; // position of the tie
    float xv,yv,zv; // velocity of the tie
    } TIE, *TIE_PTR;

// a basic 3D vector used for velocity
typedef struct VEC3D_TYP
    {
    float x,y,z; // coords of vector
    } VEC3D, *VEC3D_PTR;

// a wireframe explosion
typedef struct EXPL_TYP
    {
    int state;    // state of explosion
    int counter;   // counter for explosion
    USHORT color;  // color of explosion

    // an explosion is a collection of edges/lines
    // based on the 3D model of the tie that is exploding
    POINT3D p1[NUM_TIE_EDGES]; // start point of edge n
    POINT3D p2[NUM_TIE_EDGES]; // end point of edge n

    VEC3D  vel[NUM_TIE_EDGES]; // velocity of shrapnel
```

Playrix Ex. 1023, page 63

03 0672318350 CH01 05•02•2003 2:33 PM Page 39


```
    } EXPL, *EXPL_PTR;

// PROTOTYPES ////////////////////////////////////////////

// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// game functions
void Init_Tie(int index);

// GLOBALS ///////////////////////////////////////////////

HWND main_window_handle      = NULL; // save the window handle
HINSTANCE main_instance      = NULL; // save the instance
char buffer[256];             // used to print text

// the tie fighter is a collection of vertices connected by
// lines that make up the shape. only one tie fighter see if
// you can replicate it?

POINT3D tie_vlist[NUM_TIE_VERTS]; // vertex list for the tie fighter model
LINE3D tie_shape[NUM_TIE_EDGES]; // edge list for the tie fighter model
TIE   ties[NUM_TIES];      // tie fighters

POINT3D stars[NUM_STARS]; // the starfield

// some colors, note we can't build them until we know the bit
// depth format 5.5.5 or 5.6.5, so we wait a minute and do it in the
// Game_Init() function
USHORT rgb_green,
    rgb_white,
    rgb_red,
    rgb_blue;

// player vars
float cross_x = 0, // cross hairs
   cross_y = 0;

int cross_x_screen = WINDOW_WIDTH/2,  // used for cross hair
  cross_y_screen = WINDOW_HEIGHT/2,
```

Playrix Ex. 1023, page 64

03 0672318350 CH01 05•02•2003 2:33 PM Page 40


```
  target_x_screen = WINDOW_WIDTH/2,  // used for targeter
  target_y_screen = WINDOW_HEIGHT/2;

int player_z_vel = 4; // virtual speed of viewpoint/ship
int cannon_state = 0; // state of laser cannon
int cannon_count = 0; // laster cannon counter

EXPL explosions[NUM_EXPLOSIONS]; // the explosiions

int misses = 0; // tracks number of missed ships
int hits  = 0; // tracks number of hits
int score = 0; // take a guess :)

// music and sound stuff
int main_track_id = -1, // main music track id
  laser_id   = -1, // sound of laser pulse
  explosion_id = -1, // sound of explosion
  flyby_id   = -1; // sound of tie fighter flying by

int game_state = GAME_RUNNING; // state of game

// FUNCTIONS ////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
                  UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT  ps;      // used in WM_PAINT
HDC      hdc;    // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
    // do initialization stuff here
    return(0);
    } break;

  case WM_PAINT:
     {
```

Playrix Ex. 1023, page 65

03 0672318350 CH01 05•02•2003 2:33 PM Page 41


```
     // start painting
     hdc = BeginPaint(hwnd,&ps);

     // end painting
     EndPaint(hwnd,&ps);
     return(0);
    } break;

  case WM_DESTROY:
    {
    // kill the application
    PostQuitMessage(0);
    return(0);
    } break;

  default:break;

  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain(  HINSTANCE hinstance,
          HINSTANCE hprevinstance,
          LPSTR lpcmdline,
          int ncmdshow)
{
// this is the winmain function

WNDCLASS winclass;  // this will hold the class we create
HWND   hwnd;    // generic window handle
MSG   msg;    // generic message
HDC   hdc;    // generic dc
PAINTSTRUCT ps;   // generic paintstruct

// first fill in the window class stucture
winclass.style    = CS_DBLCLKS ¦ CS_OWNDC ¦
             CS_HREDRAW ¦ CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
```

Playrix Ex. 1023, page 66

03 0672318350 CH01 05•02•2003 2:33 PM Page 42


```
winclass.cbClsExtra  = 0;
winclass.cbWndExtra  = 0;
winclass.hInstance  = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;

// register the window class
if (!RegisterClass(&winclass))
  return(0);

// create the window, note the test to see if WINDOWED_APP is
// true to select the appropriate window flags
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
       WINDOW_TITLE,   // title
 (WINDOWED_APP ? (WS_OVERLAPPED ¦ WS_SYSMENU) : (WS_POPUP ¦ WS_VISIBLE)),
       0,0,    // x,y
       WINDOW_WIDTH, // width
            WINDOW_HEIGHT, // height
       NULL,    // handle to parent
       NULL,    // handle to menu
       hinstance,// instance
       NULL)))  // creation parms
return(0);

// save the window handle and instance in a global
main_window_handle = hwnd;
main_instance   = hinstance;

// resize the window so that client is really width x height
if (WINDOWED_APP)
{
// now resize the window, so the client area is the actual size requested
// since there may be borders and controls if this is going to be a windowed app
// if the app is not windowed then it won't matter
RECT window_rect = {0,0,WINDOW_WIDTH,WINDOW_HEIGHT};

// make the call to adjust window_rect
AdjustWindowRectEx(&window_rect,
   GetWindowStyle(main_window_handle),
```

Playrix Ex. 1023, page 67

03 0672318350 CH01 05•02•2003 2:33 PM Page 43


```
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// save the global client offsets, they are needed in DDraw_Flip()
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// now resize the window with a call to MoveWindow()
MoveWindow(main_window_handle,
      CW_USEDEFAULT, // x position
      CW_USEDEFAULT, // y position
      window_rect.right - window_rect.left, // width
      window_rect.bottom - window_rect.top, // height
      FALSE);

// show the window, so there's no garbage on first render
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed

// perform all game console specific initialization
Game_Init();

// disable CTRL-ALT_DEL, ALT_TAB, comment this line out
// if it causes your system to crash
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);

// enter main event loop
while(1)
  {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // test if this is a quit
    if (msg.message == WM_QUIT)
      break;

    // translate any accelerator keys
    TranslateMessage(&msg);

    // send the message to the window proc
    DispatchMessage(&msg);
    } // end if
```

Playrix Ex. 1023, page 68

03 0672318350 CH01 05•02•2003 2:33 PM Page 44


```
  // main game processing goes here
  Game_Main();

  } // end while

// shutdown game and release all resources
Game_Shutdown();

// enable CTRL-ALT_DEL, ALT_TAB, comment this line out
// if it causes your system to crash
SystemParametersInfo(SPI_SCREENSAVERRUNNING, FALSE, NULL, 0);

// return to Windows like this
return(msg.wParam);

} // end WinMain

// T3D II GAME PROGRAMMING CONSOLE FUNCTIONS ////////////////

int Game_Init(void *parms)
{
// this function is where you do all the initialization
// for your game

int index; // used for looping

Open_Error_File("error.txt");

// start up DirectDraw (replace the parms as you desire)
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);

// initialize directinput
DInput_Init();

// acquire the keyboard
DInput_Init_Keyboard();

// initialize directsound
DSound_Init();

// load in sound fx
explosion_id = DSound_Load_WAV("exp1.wav");
laser_id   = DSound_Load_WAV("shocker.wav");
```

Playrix Ex. 1023, page 69

03 0672318350 CH01 05•02•2003 2:33 PM Page 45


```
// initialize directmusic
DMusic_Init();

// load and start main music track
main_track_id = DMusic_Load_MIDI("midifile2.mid");
DMusic_Play(main_track_id);

// add calls to acquire other directinput devices here...

// hide the mouse
ShowCursor(FALSE);

// seed random number generator
srand(Start_Clock());

// all your initialization code goes here...

// create system colors
rgb_green = RGB16Bit(0,31,0);
rgb_white = RGB16Bit(31,31,31);
rgb_blue = RGB16Bit(0,0,31);
rgb_red  = RGB16Bit(31,0,0);

// create the starfield
for (index=0; index < NUM_STARS; index++)
  {
  // randomly position stars in an elongated cylinder stretching from
  // the viewpoint 0,0,-d to the yon clipping plane 0,0,far_z
  stars[index].x = -WINDOW_WIDTH/2 + rand()%WINDOW_WIDTH;
  stars[index].y = -WINDOW_HEIGHT/2 + rand()%WINDOW_HEIGHT;
  stars[index].z = NEAR_Z + rand()%(FAR_Z - NEAR_Z);

  // set color of stars
  stars[index].color = rgb_white;
  } // end for index

// create the tie fighter model

// the vertex list for the tie fighter
POINT3D temp_tie_vlist[NUM_TIE_VERTS] =
// color, x,y,z
{ {rgb_white,-40,40,0},  // p0
 {rgb_white,-40,0,0},  // p1
```

Playrix Ex. 1023, page 70

03 0672318350 CH01 05•02•2003 2:33 PM Page 46


```
 {rgb_white,-40,-40,0},  // p2
 {rgb_white,-10,0,0},  // p3
 {rgb_white,0,20,0},   // p4
 {rgb_white,10,0,0},   // p5
 {rgb_white,0,-20,0},  // p6
 {rgb_white,40,40,0},   // p7
 {rgb_white,40,0,0},   // p8
 {rgb_white,40,-40,0}};  // p9

// copy the model into the real global arrays
for (index=0; index<NUM_TIE_VERTS; index++)
   tie_vlist[index] = temp_tie_vlist[index];

// the edge list for the tie fighter
LINE3D temp_tie_shape[NUM_TIE_EDGES] =
// color, vertex 1, vertex 2
{ {rgb_green,0,2   },  // l0
 {rgb_green,1,3   },  // l1
 {rgb_green,3,4   },  // l2
 {rgb_green,4,5   },  // l3
 {rgb_green,5,6   },  // l4
 {rgb_green,6,3   },  // l5
 {rgb_green,5,8   },  // l6
 {rgb_green,7,9   } }; // l7

// copy the model into the real global arrays
for (index=0; index<NUM_TIE_EDGES; index++)
   tie_shape[index] = temp_tie_shape[index];

// initialize the position of each tie fighter and its velocity
for (index=0; index<NUM_TIES; index++)
  {
  // initialize this tie fighter
  Init_Tie(index);

  } // end for index

// return success
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////////
```

Playrix Ex. 1023, page 71

03 0672318350 CH01 05•02•2003 2:33 PM Page 47


```
int Game_Shutdown(void *parms)
{
// this function is where you shutdown your game and
// release all resources that you allocated

// shut everything down

// release all your resources created for the game here....

// now directsound
DSound_Stop_All_Sounds();
DSound_Shutdown();

// directmusic
DMusic_Delete_All_MIDI();
DMusic_Shutdown();

// shut down directinput
DInput_Shutdown();

// shutdown directdraw last
DDraw_Shutdown();

// return success
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////

void Start_Explosion(int tie)
{
// this starts an explosion based on the sent tie fighter

// first hunt and see if an explosion is free
for (int index=0; index < NUM_EXPLOSIONS; index++)
  {
  if (explosions[index].state==0)
    {
    // start this explosion up using the properties
    // if the tie figther index sent

    explosions[index].state  = 1; // enable state of explosion
    explosions[index].counter = 0; // reset counter for explosion
```

Playrix Ex. 1023, page 72

03 0672318350 CH01 05•02•2003 2:33 PM Page 48


```
    // set color of explosion
    explosions[index].color = rgb_green;

    // make copy of of edge list, so we can blow it up
    for (int edge=0; edge < NUM_TIE_EDGES; edge++)
      {
      // start point of edge
      explosions[index].p1[edge].x = ties[tie].x+tie_vlist
➥[tie_shape[edge].v1].x;
      explosions[index].p1[edge].y = ties[tie].y+tie_vlist
➥[tie_shape[edge].v1].y;
      explosions[index].p1[edge].z = ties[tie].z+tie_vlist
➥[tie_shape[edge].v1].z;

      // end point of edge
      explosions[index].p2[edge].x = ties[tie].x+tie_vlist
➥[tie_shape[edge].v2].x;
      explosions[index].p2[edge].y = ties[tie].y+tie_vlist
➥[tie_shape[edge].v2].y;
      explosions[index].p2[edge].z = ties[tie].z+tie_vlist
➥[tie_shape[edge].v2].z;

      // compute trajectory vector for edges
      explosions[index].vel[edge].x = ties[tie].xv - 8+rand()%16;
      explosions[index].vel[edge].y = ties[tie].yv - 8+rand()%16;
      explosions[index].vel[edge].z = -3+rand()%4;

      } // end for edge

    // done, so return
    return;
    } // end if found

  } // end for index

} // end Start_Explosion

///////////////////////////////////////////////////////////

void Process_Explosions(void)
{
// this processes all the explosions
```

Playrix Ex. 1023, page 73

03 0672318350 CH01 05•02•2003 2:33 PM Page 49


```
// loop thru all the explosions and render them
for (int index=0; index<NUM_EXPLOSIONS; index++)
  {
  // test if this explosion is active?
  if (explosions[index].state==0)
    continue;

  for (int edge=0; edge<NUM_TIE_EDGES; edge++)
    {
    // must be exploding, update edges (shrapel)
    explosions[index].p1[edge].x+=explosions[index].vel[edge].x;
    explosions[index].p1[edge].y+=explosions[index].vel[edge].y;
    explosions[index].p1[edge].z+=explosions[index].vel[edge].z;

    explosions[index].p2[edge].x+=explosions[index].vel[edge].x;
    explosions[index].p2[edge].y+=explosions[index].vel[edge].y;
    explosions[index].p2[edge].z+=explosions[index].vel[edge].z;
    } // end for edge

  // test for termination of explosion?
  if (++explosions[index].counter > 100)
    explosions[index].state = explosions[index].counter = 0;

  } // end for index

} // end Process_Explosions

//////////////////////////////////////////////////////////

void Draw_Explosions(void)
{
// this draws all the explosions

// loop thru all the explosions and render them
for (int index=0; index<NUM_EXPLOSIONS; index++)
  {
  // test if this explosion is active?
  if (explosions[index].state==0)
    continue;

  // render this explosion
  // each explosion is made of a number of edges
  for (int edge=0; edge < NUM_TIE_EDGES; edge++)
```

Playrix Ex. 1023, page 74

03 0672318350 CH01 05•02•2003 2:33 PM Page 50


```
  {
  POINT3D p1_per, p2_per; // used to hold perspective endpoints

  // test if edge if beyond near clipping plane
  if (explosions[index].p1[edge].z < NEAR_Z &&
    explosions[index].p2[edge].z < NEAR_Z)
    continue;

  // step 1: perspective transform each end point
  p1_per.x = VIEW_DISTANCE*explosions[index].p1[edge].x/explosions
➥[index].p1[edge].z;
  p1_per.y = VIEW_DISTANCE*explosions[index].p1[edge].y/explosions
➥[index].p1[edge].z;
  p2_per.x = VIEW_DISTANCE*explosions[index].p2[edge].x/explosions
➥ [index].p2[edge].z;
  p2_per.y = VIEW_DISTANCE*explosions[index].p2[edge].y/explosions
➥ [index].p2[edge].z;

  // step 2: compute screen coords
  int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
  int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
  int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
  int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

  // step 3: draw the edge
  Draw_Clip_Line16(p1_screen_x, p1_screen_y, p2_screen_x, p2_screen_y,
           explosions[index].color,back_buffer, back_lpitch);

  } // end for edge

  } // end for index

} // end Draw_Explosions

//////////////////////////////////////////////////////////

void Move_Starfield(void)
{
// move the stars

int index; // looping var
```

Playrix Ex. 1023, page 75

03 0672318350 CH01 05•02•2003 2:33 PM Page 51


```
// the stars are technically stationary,but we are going
// to move them to simulate motion of the viewpoint
for (index=0; index<NUM_STARS; index++)
  {
  // move the next star
  stars[index].z-=player_z_vel;

  // test for past near clipping plane
  if (stars[index].z <= NEAR_Z)
    stars[index].z = FAR_Z;

  } // end for index

} // end Move_Starfield

//////////////////////////////////////////////////////////

void Draw_Starfield(void)
{
// draw the stars in 3D using perspective transform

int index; // looping var

for (index=0; index<NUM_STARS; index++)
  {
  // draw the next star
  // step 1: perspective transform
  float x_per = VIEW_DISTANCE*stars[index].x/stars[index].z;
  float y_per = VIEW_DISTANCE*stars[index].y/stars[index].z;

  // step 2: compute screen coords
  int x_screen = WINDOW_WIDTH/2 + x_per;
  int y_screen = WINDOW_HEIGHT/2 - y_per;

  // clip to screen coords
  if (x_screen>=WINDOW_WIDTH || x_screen < 0 ||
    y_screen >= WINDOW_HEIGHT || y_screen < 0)
    {
    // continue to next star
    continue;
    } // end if
  else
    {
```

Playrix Ex. 1023, page 76

03 0672318350 CH01 05•02•2003 2:33 PM Page 52


```
    // else render to buffer
    ((USHORT *)back_buffer)[x_screen + y_screen*(back_lpitch >> 1)]
               = stars[index].color;
    } // end else

  } // end for index

} // Draw_Starfield

///////////////////////////////////////////////////////////

void Init_Tie(int index)
{
// this function starts a tie fighter up at the far end
// of the universe and sends it our way!

// position each tie in the viewing volume
ties[index].x = -WINDOW_WIDTH + rand()%(2*WINDOW_WIDTH);
ties[index].y = -WINDOW_HEIGHT + rand()%(2*WINDOW_HEIGHT);
ties[index].z = 4*FAR_Z;

// initialize velocity of tie fighter
ties[index].xv = -4+rand()%8;
ties[index].yv = -4+rand()%8;
ties[index].zv = -4-rand()%64;

// turn the tie fighter on
ties[index].state = 1;
} // end Init_Tie

///////////////////////////////////////////////////////////

void Process_Ties(void)
{
// process the tie fighters and do AI (what there is of it!)
int index; // looping var

// move each tie fighter toward the viewpoint
for (index=0; index<NUM_TIES; index++)
  {
  // is this one dead?
  if (ties[index].state==0)
    continue;
```

Playrix Ex. 1023, page 77

03 0672318350 CH01 05•02•2003 2:33 PM Page 53


```
  // move the next star
  ties[index].z+=ties[index].zv;
  ties[index].x+=ties[index].xv;
  ties[index].y+=ties[index].yv;

  // test for past near clipping plane
  if (ties[index].z <= NEAR_Z)
    {
    // reset this tie
    Init_Tie(index);

    // another got away
    misses++;

    } // reset tie

  } // end for index

} // Process_Ties

//////////////////////////////////////////////////////////

void Draw_Ties(void)
{
// draw the tie fighters in 3D wireframe with perspective

int index; // looping var

// used to compute the bounding box of tie fighter
// for collision detection
int bmin_x, bmin_y, bmax_x, bmax_y;

// draw each tie fighter
for (index=0; index < NUM_TIES; index++)
  {
  // draw the next tie fighter

  // is this one dead?
  if (ties[index].state==0)
    continue;

  // reset the bounding box to impossible values
  bmin_x = 100000;
```

Playrix Ex. 1023, page 78

03 0672318350 CH01 05•02•2003 2:33 PM Page 54


```
  bmax_x = -100000;
  bmin_y = 100000;
  bmax_y = -100000;

  // based on z-distance shade tie fighter
  // normalize the distance from 0 to max_z then
  // scale it to 31, so the closer the brighter
  USHORT rgb_tie_color = RGB16Bit(0,(31-31*(ties[index].z/(4*FAR_Z))),0);

  // each tie fighter is made of a number of edges
  for (int edge=0; edge < NUM_TIE_EDGES; edge++)
  {
  POINT3D p1_per, p2_per; // used to hold perspective endpoints

  // step 1: perspective transform each end point
  // note the translation of each point to the position of the tie fighter
  // that is the model is relative to the position of each tie
➥fighter -- IMPORTANT
  p1_per.x =
      VIEW_DISTANCE*(ties[index].x+tie_vlist[tie_shape[edge].v1].x)/
      (tie_vlist[tie_shape[edge].v1].z+ties[index].z);

  p1_per.y = VIEW_DISTANCE*(ties[index].y+tie_vlist[tie_shape[edge].v1].y)/
        (tie_vlist[tie_shape[edge].v1].z+ties[index].z);

  p2_per.x = VIEW_DISTANCE*(ties[index].x+tie_vlist[tie_shape[edge].v2].x)/
        (tie_vlist[tie_shape[edge].v2].z+ties[index].z);

  p2_per.y = VIEW_DISTANCE*(ties[index].y+tie_vlist[tie_shape[edge].v2].y)/
        (tie_vlist[tie_shape[edge].v2].z+ties[index].z);

  // step 2: compute screen coords
  int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
  int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
  int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
  int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

  // step 3: draw the edge
  Draw_Clip_Line16(p1_screen_x, p1_screen_y, p2_screen_x, p2_screen_y,
           rgb_tie_color,back_buffer, back_lpitch);
```

Playrix Ex. 1023, page 79

03 0672318350 CH01 05•02•2003 2:33 PM Page 55


```
// update bounding box with next edge
int min_x = min(p1_screen_x, p2_screen_x);
int max_x = max(p1_screen_x, p2_screen_x);

int min_y = min(p1_screen_y, p2_screen_y);
int max_y = max(p1_screen_y, p2_screen_y);

bmin_x = min(bmin_x, min_x);
bmin_y = min(bmin_y, min_y);

bmax_x = max(bmax_x, max_x);
bmax_y = max(bmax_y, max_y);

} // end for edge

// test if this guy has been hit by lasers???
if (cannon_state==1)
  {
  // simple test of screen coords of bounding box contain laser target
  if (target_x_screen > bmin_x && target_x_screen < bmax_x &&
    target_y_screen > bmin_y && target_y_screen < bmax_y)
   {
   // this tie is dead meat!
   Start_Explosion(index);

   // start sound
   DSound_Play(explosion_id );

   // increase score
   score+=ties[index].z;

   // add one more hit
   hits++;

   // finally reset this tie figher
   Init_Tie(index);

   } // end if

  } // end if

} // end for index
```

Playrix Ex. 1023, page 80

03 0672318350 CH01 05•02•2003 2:33 PM Page 56


```
} // end Draw_Ties

//////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for your game go here!

int    index;    // looping var

// start the timing clock
Start_Clock();

// clear the drawing surface
DDraw_Fill_Surface(lpddsback, 0);

// read keyboard and other devices here
DInput_Read_Keyboard();

// game logic here...

if (game_state==GAME_RUNNING)
{
// move players crosshair
if (keyboard_state[DIK_RIGHT])
  {
  // move cross hair to right
  cross_x+=CROSS_VEL;

  // test for wraparound
  if (cross_x > WINDOW_WIDTH/2)
   cross_x = -WINDOW_WIDTH/2;

  } // end if
if (keyboard_state[DIK_LEFT])
  {
  // move cross hair to left
  cross_x-=CROSS_VEL;

  // test for wraparound
  if (cross_x < -WINDOW_WIDTH/2)
```

Playrix Ex. 1023, page 81

03 0672318350 CH01 05•02•2003 2:33 PM Page 57


```
   cross_x = WINDOW_WIDTH/2;
  } // end if
if (keyboard_state[DIK_DOWN])
  {
  // move cross hair up
  cross_y-=CROSS_VEL;

  // test for wraparound
  if (cross_y < -WINDOW_HEIGHT/2)
   cross_y = WINDOW_HEIGHT/2;
  } // end if
if (keyboard_state[DIK_UP])
  {
  // move cross hair up
  cross_y+=CROSS_VEL;

  // test for wraparound
  if (cross_y > WINDOW_HEIGHT/2)
   cross_y = -WINDOW_HEIGHT/2;
  } // end if

// speed of ship controls
if (keyboard_state[DIK_A])
  player_z_vel++;
else
if (keyboard_state[DIK_S])
  player_z_vel--;

// test if player is firing laser cannon
if (keyboard_state[DIK_SPACE] && cannon_state==0)
  {
  // fire the cannon
  cannon_state = 1;
  cannon_count = 0;

  // save last position of targeter
  target_x_screen = cross_x_screen;
  target_y_screen = cross_y_screen;

  // make sound
  DSound_Play(laser_id);

  } // end if
```

Playrix Ex. 1023, page 82

03 0672318350 CH01 05•02•2003 2:33 PM Page 58


```
} // end if game running

// process cannon, simple FSM ready->firing->cool

// firing phase
if (cannon_state == 1)
  if (++cannon_count > 15)
   cannon_state = 2;

// cool down phase
if (cannon_state == 2)
  if (++cannon_count > 20)
   cannon_state = 0;

// move the starfield
Move_Starfield();

// move and perform ai for ties
Process_Ties();

// Process the explosions
Process_Explosions();

// lock the back buffer and obtain pointer and width
DDraw_Lock_Back_Surface();

// draw the starfield
Draw_Starfield();

// draw the tie fighters
Draw_Ties();

// draw the explosions
Draw_Explosions();

// draw the crosshairs

// first compute screen coords of crosshair
// note inversion of y-axis
cross_x_screen = WINDOW_WIDTH/2 + cross_x;
cross_y_screen = WINDOW_HEIGHT/2 - cross_y;
```

Playrix Ex. 1023, page 83

03 0672318350 CH01 05•02•2003 2:33 PM Page 59


```
// draw the crosshair in screen coords
Draw_Clip_Line16(cross_x_screen-16,cross_y_screen,
         cross_x_screen+16,cross_y_screen,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen,cross_y_screen-16,
         cross_x_screen,cross_y_screen+16,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen-16,cross_y_screen-4,
         cross_x_screen-16,cross_y_screen+4,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen+16,cross_y_screen-4,
         cross_x_screen+16,cross_y_screen+4,
         rgb_red,back_buffer,back_lpitch);

// draw the laser beams
if (cannon_state == 1)
  {
  if ((rand()%2 == 1))
   {
   // right beam
   Draw_Clip_Line16(WINDOW_WIDTH-1, WINDOW_HEIGHT-1,
       -4+rand()%8+target_x_screen,-4+rand()%8+target_y_screen,
       RGB16Bit(0,0,rand()),back_buffer,back_lpitch);
   } // end if
  else
   {
   // left beam
   Draw_Clip_Line16(0, WINDOW_HEIGHT-1,
       -4+rand()%8+target_x_screen,-4+rand()%8+target_y_screen,
       RGB16Bit(0,0,rand()),back_buffer,back_lpitch);
   } // end if

  } // end if

// done rendering, unlock back buffer surface
DDraw_Unlock_Back_Surface();

// draw the informtion
sprintf(buffer, "Score %d   Kills %d   Escaped %d", score, hits, misses);
Draw_Text_GDI(buffer, 0,0,RGB(0,255,0), lpddsback);
```

Playrix Ex. 1023, page 84

03 0672318350 CH01 05•02•2003 2:33 PM Page 60


```
if (game_state==GAME_OVER)
  Draw_Text_GDI("G A M E O V E R", 320-8*10,240,RGB(255,255,255), lpddsback);

// check if the music has finished, if so restart
if (DMusic_Status_MIDI(main_track_id)==MIDI_STOPPED)
  DMusic_Play(main_track_id);

// flip the surfaces
DDraw_Flip();

// sync to 30ish fps
Wait_Clock(30);

// check for game state switch
if (misses > 100)
  game_state = GAME_OVER;

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) ¦¦ keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if
// return success
return(1);
} // end Game_Main

////////////////////////////////////////////////////////////
```

Pretty short for a 3D game, huh! That's an entire 3D Win32/DirectX game. Well, almost—there's a few thousand lines of code in the `T3DLIB` modules, but we'll just pretend that it's like DirectX and someone else wrote it.

Before we move on to analyzing the code, I want you to compile this bad boy yourself. You are not allowed to move on until you get a successful compile! Fair enough? So, follow the steps that we outlined as far as setting up the compiler to create a Win32 .EXE application with all the search paths and link lists ready for DirectX. Then, once you have your project ready, you must include the source files:

```
T3DLIB1.CPP, T3DLIB2.CPP, T3DLIB3.CPP, RAIDERS3D.CPP
```

Of course, the header files

```
T3DLIB1.H, T3DLIB2.H, T3DLIB3.H
```

Playrix Ex. 1023, page 85

03 0672318350 CH01 05•02•2003 2:33 PM Page 61

must be in the working directory of the compiler. And finally, you need to make absolutely sure you have included the DirectX .LIB files in the project along with the .CPP files OR in the link list. You will only need the following DirectX .LIB files:

```
DDRAW.LIB, DSOUND.LIB, DINPUT.LIB, DINPUT8.LIB
```

You can call the .EXE anything you like—maybe `TEST.EXE` or `RAIDERS3D_TEST.EXE`—but don't move on until you can compile.

The Event Loop

`WinMain()` is the main entry point for all Windows programs, just like `main()` is the entry point for all DOS/Unix programs (please wash your mouth out if you said Unix out loud). In any case, the `WinMain()` for *Raiders3D* creates a window and then enters right into the event loop. If Windows needs to do something, it does it. In any case, `WinMain()` begins by creating a Windows class and registering it. Next the game window is created, then a call is made to our function `Game_Init()` that handles all the initialization of the game. When all the initialization is complete, a standard Windows event loop is entered that "peeks" for a message. If a message is found, the Windows procedure, or *WinProc* is called, which handles it. Otherwise, a call to our main game programming function `Game_Main()` is called. This is where the real action occurs for our game.

NOTE

Readers from the previous *Tricks* will notice that there is some extra code in `WinMain()`'s initialization to handle windowed graphics and the resizing of the window. This new feature, along with 16-bit support, is part of the revised `T3DLIB` game engine. However, the majority of the 3D for this book will still support 8-bit graphics, because getting speed out of 16-bit software 3D is still too slow in general.

If you want to, you could loop in `Game_Main()` forever, never releasing back to the main event loop in `WinMain()`. This would be bad because Windows would never receive any messages, and you would starve the system. What we need to do is perform one frame of animation and logic and then return back to `WinMain()`. This way, Windows will continue to function and process messages. This process is shown in Figure 1.17.

The Core 3D Game Logic

Once in `Game_Main()`, the logic for *Raiders3D* executes, and the 3D game image is rendered into an offscreen workspace (the double buffer, or in DirectX lingo, "back buffer"). It is finally displayed on the physical display at the end of the loop via the call to `DDraw_Flip()`, which creates the illusion of animation. The game loop consists of the standard sections as defined previously in the elements of a 2D/3D game. The area that I want to focus on is the 3D graphics.

Playrix Ex. 1023, page 86

03 0672318350 CH01 05•02•2003 2:33 PM Page 62

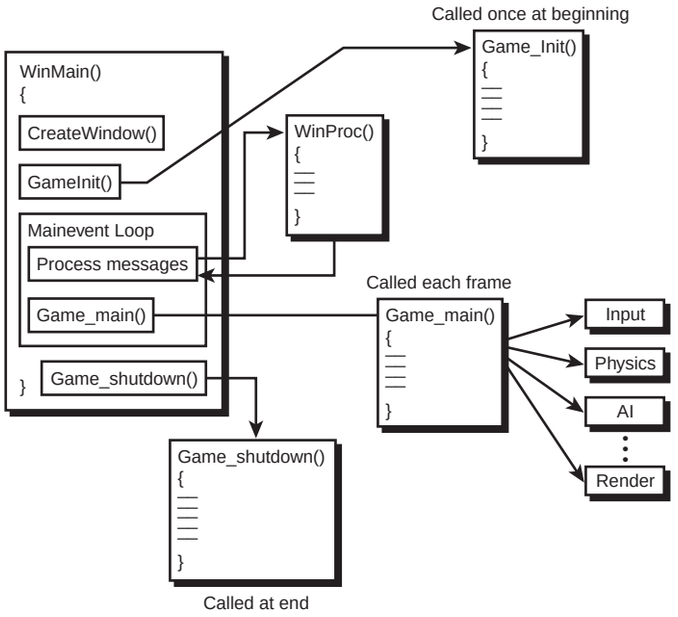


FIGURE 1.17 The entire message-processing system.

The game logic for the enemy AI is fairly simple. An alien ship is generated at a random position in 3D space at a distance just beyond view. Figure 1.18 shows the 3D space of the *Raiders3D* universe. As you can see, the camera or viewpoint is fixed along the negative z-axis at a position (0,0,-zd), where –zd is the viewing distance from the virtual window that the image is projected on. The 3D space is called a left-handed system because the positive z-axis goes into the screen.

After an enemy ship is generated, it follows a trajectory vector along a fixed path that intersects the player's viewing position—thus it's on a collision course, more or less. The vector and initial position are generated in the function `Init_Tie()`. Your goal as the player is to target the enemy and fire—that's it.

So, the question is, how is the 3D display generated? The enemy ships are nothing more than polygon objects (collections of lines) that outline the shape of the 3D object. That is, they are 2D outlines, rather than full 3D objects. The key to all 3D is perspective. Figure 1.19 shows the difference between an orthographic-projected house and a perspective-projected house. These are the two basic types of projections used in 3D graphics systems.

Playrix Ex. 1023, page 87

03 0672318350 CH01 05•02•2003 2:33 PM Page 63

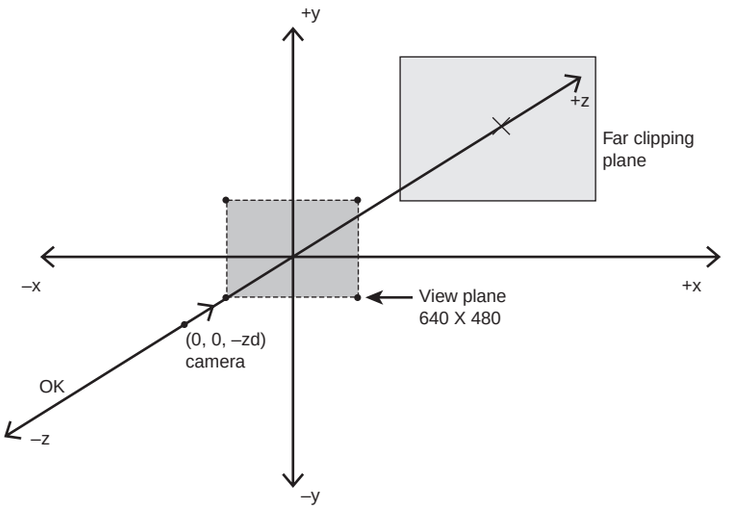


FIGURE 1.18 The universe of *Raiders3D*.

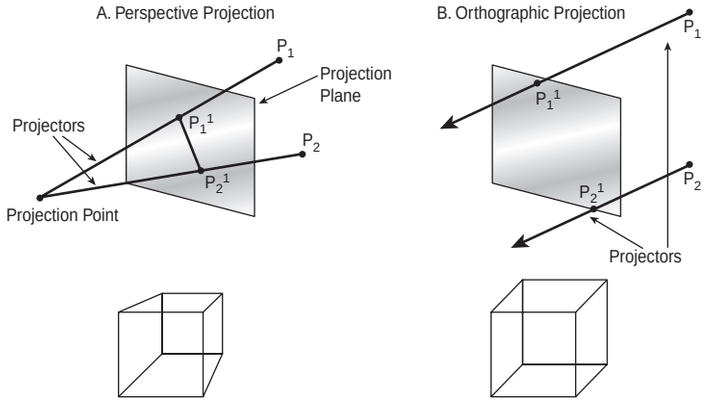


FIGURE 1.19 Orthographic versus perspective projections.

3D Projections

The orthographic projection is good for technical drawings and images where perspective warping is not desired. The math for an orthographic projection is very simple—basically, the z coordinates of each point are simply thrown away, as shown below:

Playrix Ex. 1023, page 88

03 0672318350 CH01 05•02•2003 2:33 PM Page 64

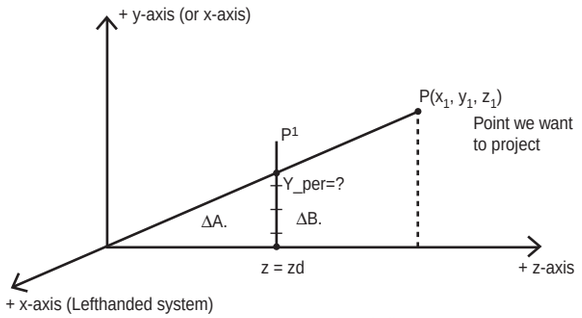
EQUATION 1.1 The Orthographic Projection

Given (x,y,z) in 3D space:

$x_{ortho} = x$

$y_{ortho} = y$

The perspective projection is a little complex, and at this point I don't want to get too much into its derivation. In general, we want to take the z coordinate into consideration along with the viewing distance to arrive at a projected (x_{per}, y_{per}) in 2D that can be plotted on the 2D screen. The mathematics are based on the image shown in Figure 1.20, which is based on a similar triangles argument. It can be shown that the point (x_{per}, y_{per}) can be calculated with the following equations.



Using similar Δs we see that ΔA is similar to ΔB, therefore:

$$\frac{Y\_per}{Y_1} = \frac{zd}{z}$$

$$Y\_per = Y_1 \cdot \frac{zd}{z}$$

FIGURE 1.20 The derivation of the perspective transformation.

EQUATION 1.2 The Perspective Projection

Given (x,y,z) and the viewing distance `zd` in 3D space:

$x_{per} = zd*x/z$

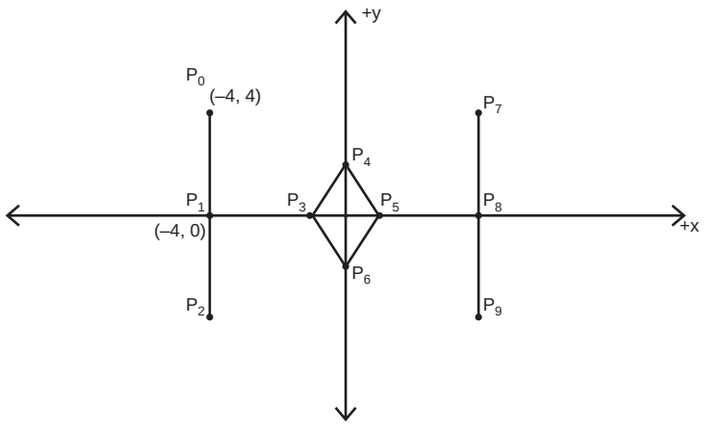
$y_{per} = zd*y/z$

With that simple equation, we can move the polygon objects in 3D just like 2D objects, use the perspective projection equations to warp the points before rendering, and then the objects will look and move correctly in 3D. Of course, there are some details about converting the final coordinates (x_{per}, y_{per}) to actual screen coordinates, but that's not

Playrix Ex. 1023, page 89

03 0672318350 CH01 05•02•2003 2:33 PM Page 65

important right now. All you need to know is that 3D is nothing more than objects represented in 3D that move in 3D, and that they are projected from 3D onto a 2D viewing surface (the screen) with mathematics that take perspective into consideration.



	Vertex List P_0... P_9
P_0	(–4, 4)
P_1	(–4, 0)
P_2	(–4, –4)
P_3	(–1, 0)
P_4	(0, 2)
P_5	(1, 0)
P_6	(0, –2)
P_7	(4, 4)
P_8	(4, 0)
P_9	(4, –4)

	Edge List
edge 0	P_0–P_2
edge 1	P_1–P_3
edge 2	P_3–P_4
edge 3	P_4–P_5
edge 4	P_5–P_6
edge 5	P_6–P_3
edge 6	P_5–P_8
edge 7	P_7–P_9

FIGURE 1.21 The wireframe model of the TIE fighter.

Based on this knowledge, each enemy ship is transformed with the perspective projection and rendered onto the screen. Figure 1.21 shows a wireframe model on graph paper of the TIE fighter enemy. Basically, this model is used as the basis for the rendering. A table of the actual positions of each TIE fighter is kept in an array of the following structures:

```
// a tie fighter
typedef struct TIE_TYP
    {
    int state;   // state of the tie, 0=dead, 1=alive
    float x, y, z; // position of the tie
    float xv,yv,zv; // velocity of the tie
    } TIE, *TIE_PTR;
```

Playrix Ex. 1023, page 90

03 0672318350 CH01 05•02•2003 2:33 PM Page 66

When it's time to draw the enemy fighters, the data in the form of a TIE structure and the TIE 3D model are used along with the perspective projection. Presto—a moving 3D image!

You'll also notice that the TIE fighters get brighter as they get closer. This is an easy effect to implement. Basically, the z-axis distance is used as a scaling factor to control the brightness of the TIE fighter as it's drawn.

The Star Field

The star field is nothing more than a collection of single points generated from an emitter out in space. Furthermore, they are recycled as they pass the viewpoint of the player and go behind him. The points are rendered as full 3D objects with the perspective transform, but have a size of 1×1×1 pixel, so they are true "points" and always look like single pixels.

The Laser Cannons and Collision Detection

The laser cannons that the player fires are nothing more than 2D lines projected from the corners of the screen that converge on the crosshairs. The collision detection is accomplished by looking at the 2D projection of all the 3D ships on the viewing screen and testing if the lasers are contained within the bounding box of each projection. This is shown graphically in Figure 1.22.

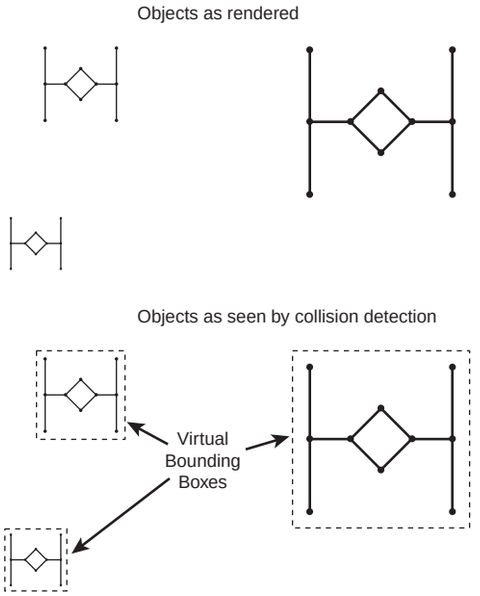


FIGURE 1.22 Using the 2D projected bounding box for collision detection.

Playrix Ex. 1023, page 91

03 0672318350 CH01 05•02•2003 2:33 PM Page 67

This algorithm works because lasers travel at the speed of light. It doesn't matter if the target is 10 meters or 10,000 km away—if you target it with a laser beam and the laser beam intersects even the projection of the 3D image, there is a collision.

The Explosions

The explosions for this game are fairly impressive for the amount of code. When a TIE is hit by a laser pulse, I copied the lines that make up the 3D model of the TIE into a secondary data structure. The lines are then moved apart randomly in 3D, as if they were pieces of the TIE. The pieces move for a couple of seconds, and then the explosion terminates. The effect is very realistic, and is implemented in less than 100 lines of code.

Playing *Raiders3D*

To play the game, simply click on `RAIDERS3D.EXE` on the CD and the program will launch immediately. The controls are as follows:

Arrow keys	Move crosshair
Space bar	Fire lasers
Esc	Exits game

Note: If 100 enemies get by you, the game is over!

The game uses DirectDraw, DirectInput, DirectSound, and DirectMusic, so make sure that your runtime DirectX system is operational.

NOTE

At the end of the book, just for fun we will rewrite *Raiders3D* with the fully lit, full 3D-modeled, textured polygon engine, just to see the difference!

Summary

This chapter has been nothing more than a get-your-feet-wet introduction to 3D game programming. The most important thing to get out of it is compiling DirectX programs, if nothing else! However, we did cover game programming, game loops, general 3D perspective transforms, and using the compiler. You were also introduced to the `T3DLIB` game engine from the previous *Tricks* that we will use to help get a DirectX system up and running, so we can focus on 3D game programming and not on creating surfaces, querying for interfaces, and loading sound FX! Now that you can compile, try messing with the game—add more aliens, some asteroids, and whatever else—and meet me in the next chapter …

Playrix Ex. 1023, page 92

CHAPTER **2**

A Crash Course in Windows and DirectX

"No, Lieutenant, your men are already dead."

—Agent Smith, *The Matrix*

IN THIS CHAPTER

- The Win32 Programming Model
- The Bare Minimum for a Windows Program
- A Basic Windows Application
- DirectX and COM Crash Course
- Flash Introduction to COM

In this chapter, we are basically going to take a crash course in Windows and DirectX programming. This book mostly assumes that you know Win32 and DirectX programming. However, I tried to write the book so that if you don't know Win32/DirectX programming, you can simply use the APIs that I create and focus on the 3D part. However, for those that don't have the pleasure of knowing Win32/DirectX programming, let me indulge you just a bit. Here's what we'll cover:

- Windows programming
- Event loops
- Writing a simple Windows shell
- DirectX fundamentals
- Component object model

The Win32 Programming Model

Windows is a multitasking/multithreaded O/S, but it's also an *event-driven* O/S. Unlike most DOS programs (all, actually), most Windows programs sit and wait for the user to do something (which fires an event) and then Windows responds to the event and takes action. Take a look at Figure 2.1 to see this graphically. Figure 2.1 depicts a number of application windows, each sending their events or messages to Windows

Playrix Ex. 1023, page 94

04 0672318350 CH02 05•02•2003 2:33 PM Page 70

to be processed. Windows does some of the processing, but the majority of the messages or events are passed through to your application program for processing.

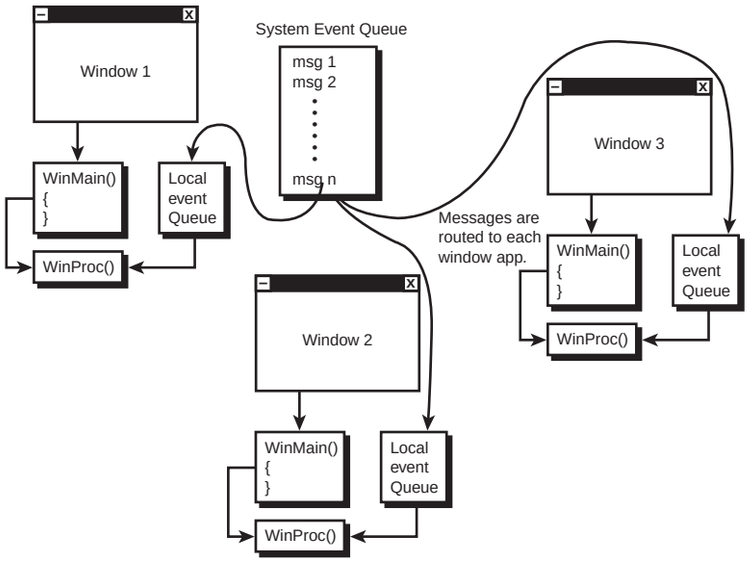


FIGURE 2.1 Windows event handling.

The good news is that for the most part you don't need to concern yourself with other applications that are running—Windows will handle them for you. All you have to worry about is your own application and the processing of messages for your application's window(s). This wasn't the entire truth in Windows 3.0/3.1. These versions of Windows weren't true multitasking O/Ss, and each application had to yield to the next. This made for a rather rough or sluggish feel for applications running under these versions. If other applications were hogging the system, there wasn't anything that compliant applications could do. However, with Windows *9x*/Me/2000/XP/NT, this isn't the case. The O/S will pull the rug from under your application whenever it feels like it—of course, it pulls it so quickly that you'll never notice!

NOTE

At this point, you know all you need to about O/S concepts. Luckily, Windows is such a nice O/S to write games for these days that you won't have to worry about scheduling and so on. All you need to worry about is the game code and pushing the machine to its limits.

Playrix Ex. 1023, page 95

04 0672318350 CH02 05•02•2003 2:33 PM Page 71

The Bare Minimum for a Windows Program

Now that you have a general overview of the Windows O/S and some of its properties and underlying design, let's begin our crash course into Windows programming with our first Windows program.

It's customary to write a "Hello World" program in any new language or O/S that you're learning, so we'll do the same. The following listing is the standard DOS-based "Hello World."

```
// DEMOII2_1.CPP - hello world standard version
#include <stdio.h>

// main entry point for all standard DOS/console programs
void main(void)
{
printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // end main
```

Now let's see how it's done with Windows.

> **NOTE**
>
> By the way, if you want to compile `DEMOII2_1.CPP`, you can actually create what's called a *console application* with the Visual C++ or Borland compilers. These are like DOS applications, but 32-bit. They only run in text mode, but are great for testing out ideas and algorithms.

It All Begins with `WinMain()`

All Windows programs begin execution at the function named `WinMain()`. This is equivalent to `main()` in a straight DOS/Unix program. What you do in `WinMain()` is up to you. If you want, you can create a window, start processing events, draw images on the screen and so on. On the other hand, you can just make a call to one of the hundreds (or thousands) of Win32 API functions. This is what we're going to do next.

As an example, I just want to print something on the screen in a little message box. There just so happens to be a Win32 API function that does just that—`MessageBox()`. The following listing is a complete, compilable Windows program that creates and displays a message box that you can move around and close if you want.

```
// DEMOII2_2.CPP - a simple message box
#define WIN32_LEAN_AND_MEAN // no MFC

#include <windows.h>    // the main windows headers
#include <windowsx.h>    // a lot of cool macros
```

Playrix Ex. 1023, page 96

04 0672318350 CH02 05•02•2003 2:33 PM Page 72


```
// main entry point for all windows programs
int WINAPI WinMain(HINSTANCE hinstance,
          HINSTANCE hprevinstance,
       LPSTR lpcmdline,
       int ncmdshow)
{
// call message box api with NULL for parent window handle
MessageBox(NULL, "THERE CAN BE ONLY ONE!!!",
         "MY FIRST WINDOWS PROGRAM",
         MB_OK ¦ MB_ICONEXCLAMATION);

// exit program
return(0);

} // end WinMain
```

To compile the program, follow these steps:

1. Create a new Win32 .EXE Application Project and include `DEMOII2_2.CPP` from `T3DIICHAP02\` on the CD-ROM.
2. Compile and link the program.
3. Run it! (Or, run the precompiled version `DEMOII2_2.EXE` on the CD.)

And you thought that a basic Windows program had hundreds of lines of code! Anyway, when you compile and run the program, you should see something like what's depicted in Figure 2.2.



FIGURE 2.2 Screenshot of `DEMOII2_2.EXE` running.

Now that you have a complete Windows program, let's take it apart line by line and see what's going on.

The very first line of code is

```
#define WIN32_LEAN_AND_MEAN
```

This deserves a bit of explanation. There are two ways to write Windows programs: with *MFC (The Microsoft Foundation Classes)*, or with the *SDK (Software Development Kit)*. MFC is

Playrix Ex. 1023, page 97

04 0672318350 CH02 05•02•2003 2:33 PM Page 73

much more complex, based totally on C++ and classes, and it is 10 times more overkill than you will ever need for games. The SDK, on the other hand, is manageable, can be learned in a week or two (at least the rudiments of it), and uses straight C. Hence, the SDK is what I'm going to use in this book.

Back to the explanation: `WIN32_LEAN_AND_MEAN` instructs the compiler (header file logic, actually) to not include extraneous MFC overhead.

Next the following header files are included:

```
#include <windows.h>
#include <windowsx.h>
```

The first include of `windows.h` really includes all the Windows header files—and there are a lot of them, so this is like an inclusion macro with nested includes to save you from manually including dozens of explicit header files.

The second include `windowsx.h` is the macro and constant header that contains a number of important macros and constants that make Windows programming easier.

And now for the important part—the main entry point of all Windows applications, `WinMain()`:

```
int WINAPI WinMain(HINSTANCE hinstance,
         HINSTANCE hprevinstance,
       LPSTR lpcmdline,
       int ncmdshow);
```

First off, you should notice that weird `WINAPI` declarator. This is equivalent to the PASCAL function declarator, which forces the parameters to be passed from left to right, rather than the normal right-to-left order with the default `CDECL`. However, the PASCAL calling convention declarator is now obsolete, and `WINAPI` has taken its place. You ***must*** use `WINAPI` for the `WinMain()` function; otherwise, the startup code will end up passing the parameters incorrectly to the function!

Next let's look at each of the parameters in detail.

- `hinstance`—This parameter is the instance handle that Windows generates for your application. *Instances* are pointers or numbers used to track resources. In this case, `hinstance` is used to track your application, like a name or address. When your application is executed, Windows will supply this parameter to your application.
- `hprevinstance`—This parameter is no longer used, but in previous versions of Windows, it tracked the previous instance of the application. In other words, the instance of the application that launched the current one. No wonder Microsoft got rid of it! It's like time travel—it gives me a headache thinking about it.

Playrix Ex. 1023, page 98

04 0672318350 CH02 05•02•2003 2:33 PM Page 74

- `lpcmdline`—A `NULL`-terminated string similar to the command-line parameters of the standard C/C++ `main(int argc, char **argv)` function, except that there isn't a separate parameter analogous to `argc` indicating the number of command-line parameters. For example, if you create a Windows application called `TEST.EXE` and launch it with the following parameters:

  ```
  TEST.EXE one two three
  ```

  then `lpcmdline` will contain the following data:

  ```
  lpcmdline = "one two three"
  ```

  Notice that the name of the .EXE itself is *not* part of the command line.

- `ncmdshow`—This final parameter is simply an integer that is passed to the application during launching, indicating how the main application window is to be opened. Thus, the user has a little control over how the application starts up. Of course, you as the programmer can disregard this if you want, but it's there if you want to use it (you pass it to `ShowWindow()`, but we're getting ahead of ourselves). Table 2.1 contains a list of the most common values `ncmdshow` can take on.

TABLE 2.1 Windows Codes for `ncmdshow`

Value	Meaning
`SW_SHOWNORMAL`	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when displaying the window for the first time.
`SW_SHOW`	Activates the window and displays it in its current size and position.
`SW_HIDE`	Hides the window and activates another window.
`SW_MAXIMIZE`	Maximizes the specified window.
`SW_MINIMIZE`	Minimizes the specified window and activates the next top-level window in the Z order.
`SW_RESTORE`	Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized window.
`SW_SHOWMAXIMIZED`	Activates the window and displays it as a maximized window.
`SW_SHOWMINIMIZED`	Activates the window and displays it as a minimized window.
`SW_SHOWMINNOACTIVE`	Displays the window as a minimized window. The active window remains active.
`SW_SHOWNA`	Displays the window in its current state. The active window remains active.
`SW_SHOWNOACTIVATE`	Displays a window in its most recent size and position. The active window remains active.

As you can see from Table 2.1, there are a lot of settings for `ncmdshow` (many of which make no sense at this point). In reality, the majority of them will never be sent in

Playrix Ex. 1023, page 99

04 0672318350 CH02 05•02•2003 2:33 PM Page 75

ncmdshow; you will use them with another function ShowWindow(), which actually displays a window after it's created. However, we will get to this a little later in the chapter.

The point I want to make is that Windows has a lot of options, flags, and so on that you will never use, but they are still there. It's like VCR programming options—more is always better, as long as you don't need to use them if you don't want to. Windows is designed this way. It has to make everybody happy, so that means a lot of options. In fact, we will use SW_SHOW, SW_SHOWNORMAL, and SW_HIDE 99% of the time.

Finally, let's talk about the actual function call to MessageBox() within WinMain(). This does all the work for us. MessageBox() is a Win32 API function that does something useful for us, so we don't have to do it. In the case of MessageBox(), it is used to display messages with various icons along with a button or two. You see, displaying simple messages is so common in Windows applications that a function was written just to save application programmers the half-hour or so needed to write one every time.

MessageBox() doesn't do much, but it does enough to get a window up on the screen, ask a question, and wait for the user's input. The following is the prototype for MessageBox():

```
int MessageBox( HWND  hwnd,   // handle of owner window
        LPCTSTR lptext,  // address of text in message box
        LPCTSTR lpcaption,// address of title of message box
        UINT  utype);  // style of message box
```

- hwnd—This is the handle of the window you want the message box attached to. At this point, we haven't covered window handles, so just think of it as the parent of the message box. In the case of DEMOII2_2.CPP, we are setting it to NULL, which means use the Windows Desktop as the parent window.
- lptext—This is a NULL-terminated string containing the text you want displayed.
- lpcaption—This is a NULL-terminated string containing the caption for the window of the message box dialog.
- utype—This is about the only exciting parameter of the bunch. It controls what kind of message box is displayed. Take a look at Table 2.2 to see a list (somewhat abridged) of the various options.

TABLE 2.2 MessageBox() Options

Flag	Meaning
The following settings group control the general style of message box.	
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.

Playrix Ex. 1023, page 100

04 0672318350 CH02 05•02•2003 2:33 PM Page 76

TABLE 2.2 Continued

Flag	Meaning
The following settings group control the general style of message box.	
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
This group controls the addition of a graphic icon to add a little poor man's multimedia.	
MB_ICONEXCLAMATION	An exclamation point icon appears in the message box.
MB_ICONINFORMATION	An icon consisting of a lowercase letter *i* in a circle appears in the message box.
MB_ICONQUESTION	A question mark icon appears in the message box.
MB_ICONSTOP	A stop sign icon appears in the message box.
And finally, this flag group controls which button is highlighted by default.	
MB_DEFBUTTON*n*	Where *n* is a number (1…4) indicating which button is the default, numbered from left to right.

(Note: There are additional advanced O/S level flags, but we aren't concerned with them. You can always look them up in the online compiler Win32 SDK Help if you want to know more.)

You can logically `OR` the values together in Table 2.2 to create the desired message box. Usually, you will only `OR` one flag from each group.

And of course, like all good Win32 API functions, `MessageBox()` returns a value to let you know what happened. In our case, who cares, but in general, you might want to know the return value if the message box was a yes/no question, and so forth. The possible return values are listed in Table 2.3.

TABLE 2.3 Return Values for `MessageBox()`

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDYES	Yes button was selected.

Finally, a table where I can list all the values without defoliating an entire forest! Anyway, this completes the line-by-line analysis of our first Windows program—click!

Now I want you to get comfortable making changes to the program and compiling it in different ways. Try altering the various compiler options, like optimization, code

Playrix Ex. 1023, page 101

04 0672318350 CH02 05•02•2003 2:33 PM Page 77

generation, and so on. Then try running the program through the debugger and see whether you can figure that out. When you're done, come back. I'm going to get something to eat—I'm ready to pass out ….

If you want to hear some sound, a cheap trick is to use the `MessageBeep()` function. You can look it up in the Win32 SDK. It's similar to the `MessageBox()` function as far as simplicity of use. Here it is:

```
BOOL MessageBeep(UINT utype); // the sound to play
```

where the different sounds can be one of the constants shown in Table 2.4.

TABLE 2.4 Sound Identifiers for `MessageBeep()`

Value	Meaning
`MB_ICONASTERISK`	System asterisk sound.
`MB_ICONEXCLAMATION`	System exclamation sound.
`MB_ICONHAND`	System hand sound.
`MB_ICONQUESTION`	System question sound.
`MB_OK`	System default sound.
`0xFFFFFFFF`	Standard beep using the computer speaker—yuck!

(Note: if you have an MS-Plus theme installed, you're sure to get some interesting results.)

See how cool the Win32 API is? There are literally hundreds of functions to play with. Granted, they aren't the fastest things in the world, but for general housekeeping, I/O, and GUI stuff, the Win32 API is very convenient.

Let's take a moment to summarize what we know at this point about Windows programming. The first thing is that Windows is multitasking/multithreaded, so multiple applications can run at once. However, we don't have to do anything to make this happen. What does concern us is that Windows is event-driven. This means that we have to process events (which we have no idea how to do at this point) and respond to them. Okay, sounds good. Finally, all Windows programs start with the function `WinMain()`, which has a few more parameters than the normal DOS `main()`, but is within the realm of logic and reason.

With all that in mind, it's time to write a very basic Windows application that will in fact be the basis for the game engine template we will write later in the chapter.

A Basic Windows Application

Because the goal of this book is to write 3D games that run on Windows, we don't need to know much about Windows programming. Actually, all you need is a basic skeleton Windows program that opens a window, processes messages, and calls the main game loop—that's it. My goal in this section is to first teach you how to create simple Windows

Playrix Ex. 1023, page 102

04 0672318350 CH02 05•02•2003 2:33 PM Page 78

programs, but at the same time to lay the groundwork for a game programming "shell" application that looks like a 32-bit DOS/Unix machine. So let's get started.

The main point of any Windows program is to open a window. A Window is nothing more that a workspace that displays information such as text and graphics with which the user can interact. To create a fully functional Windows program, you only have to follow these steps:

- Create a Windows class.
- Create an event handler, or `WinProc`, as it's called.
- Register the Windows class with Windows.
- Create a window with the previously created Windows Class.
- Create a main event loop that retrieves and dispatches Windows messages to the event handler.

Let's take a look at each step in detail.

The Windows Class

Windows is really an object-oriented O/S, and therefore there are a lot of concepts and procedures in Windows that have their roots in C and C++—one of which is *Window Classes*. Each window, control, list box, dialog box, gadget, and so forth in Windows is actually a window. What makes them all different is the "Class" that defines them. A Windows Class is a description of a window type that Windows can handle.

There are a number of predefined Window Classes, such as buttons, list boxes, file selectors, and so on. However, you are free to create your own Window Classes. In fact, you will create at least one Window Class for each application you write. Otherwise, your program would be rather boring. So you can think of the Window Class as a template for Windows to follow when drawing your window, as well as processing messages for it.

There are two data structures available to hold Windows Class information: `WNDCLASS` and `WNDCLASSEX`. `WNDCLASS` is the older of the two, and will probably be obsolete soon, so we will use the new extended version `WNDCLASSEX`. The structures are very similar, and if you are interested, you can look up the old `WNDCLASS` in the Win32 Help. Anyway, let's take a look at `WNDCLASSEX` as defined in the Windows header files:

```
typedef struct _WNDCLASSEX
    {
    UINT  cbSize;    // size of this structure
    UINT  style;     // style flags
    WNDPROC lpfnWndProc;  // function pointer to handler
    int   cbClsExtra;  // extra class info
    int   cbWndExtra;  // extra window info
    HANDLE hInstance;   // the instance of the application
```

Playrix Ex. 1023, page 103

04 0672318350 CH02 05•02•2003 2:33 PM Page 79


```
HICON  hIcon;     // the main icon
HCURSOR hCursor;    // the cursor for the window
HBRUSH hbrBackground; // the background brush to paint the window
LPCTSTR lpszMenuName; // the name of the menu to attach
LPCTSTR lpszClassName; // the name of the class itself
HICON  hIconSm;    // the handle of the small icon
} WNDCLASSEX;
```


So what you do is create one of these structures and then fill in all the fields.

```
WNDCLASSEX winclass; // a blank windows class
```

Let's see how to fill in each field now.

The first field `cbSize` is very important. It is the size of the `WNDCLASSEX` structure itself. You might be wondering why the structure needs to know how big it is. That's a good question. The reason is that if this structure is passed as a pointer, the receiver can always check the first field (4 bytes) to decide how long the data chunk is at the very least, and jump over it to get to the end. It's like a precaution and a little helper info, so other functions don't have to compute the class size during runtime. Therefore, all you have to do is set it up:

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

The next field deals with the style information flags that describe the general properties of the window. There are a lot of these flags, so I'm not going to show them all, but suffice it to say that you can create any type of window with them. A good working subset of the possible flags is shown in Table 2.5. You can logically `OR` these values together to derive the type of window you want.

TABLE 2.5 Style Flags for Window Classes

Flag	Meaning
`CS_HREDRAW`	Redraws the entire window if a movement or size adjustment changes the width of the window.
`CS_VREDRAW`	Redraws the entire window if a movement or size adjustment changes the height of the window.
`CS_OWNDC`	Allocates a unique device context for each window in the class (more on this later).
`CS_DBLCLKS`	Sends double-click messages to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
`CS_PARENTDC`	Sets the clipping region of the child window to that of the parent window so that the child can draw on the parent.
`CS_SAVEBITS`	Saves the client image in a window, so you don't have to redraw it every time the window is obscured, moved, and so on. However, this takes up more memory and is slower than doing it yourself.
`CS_NOCLOSE`	Disables the Close command on the System menu.

(Note: the most commonly used flags are highlighted.)

Playrix Ex. 1023, page 104

04 0672318350 CH02 05•02•2003 2:33 PM Page 80

There are a lot of flags in Table 2.5, and I can't blame you if you're confused. For now though, we'll just set the style flags to indicate that we want the window redrawn if it is moved or resized, and we want a static device context along with the capability to handle double-click events.

Device contexts are used for GDI graphics rendering into a window. Hence, if you want to do graphics, you need to request a device context for the particular window you are interested in. If we set the Windows Class such that it has its own device context via `CS_OWNDC`, we can save some time instead of requesting one all the time. Did that help at all, or did I make it worse? Windows is like that—the more you know, the more you don't. Anyway, here's how to set the `style` field to make a window that suits our needs:

```
winclass.style = CS_VREDRAW ¦ CS_HREDRAW ¦ CS_OWNDC ¦ CS_DBLCLICKS;
```

The next field of the `WNDCLASSEX` structure `lpfnWndProc` is a function pointer to the event handler. Basically, what we are setting here is a *callback* function for the class. Callback functions are fairly common in Windows programming. A callback function works like this: When something happens, instead of you randomly polling for it, Windows notifies you by calling a callback function you supply. Then within the callback function, you take whatever action needs to be taken.

This metaphor is how the basic Window event loop and event handler works. You supply a callback function to the Windows Class (of course, with a specific prototype), and when an event occurs, Windows calls it for you. This is shown graphically in Figure 2.3. Again, we will cover this more in later sections. For now, let's just set it to the event function that we'll write in a moment.

```
winclass.lpfnWndProc = WinProc; // this is our function
```

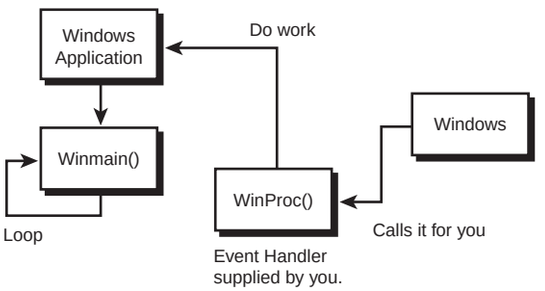


FIGURE 2.3 The Windows event handler callback in action.

Playrix Ex. 1023, page 105

04 0672318350 CH02 05•02•2003 2:33 PM Page 81

NOTE

If you're not familiar with function pointers, they are like virtual function in C++. If you're not familiar with virtual functions, I guess I have to explain it <BG>. Suppose you have two functions that operate on two numbers:

```
int Add(int op1, int op2) {return(op1+op2);}
int Sub(int op1, int op2) {return(op1-op2);}
```

You want to be able to call either function with the same call—this can be done with a function pointer:

```
// define a function pointer that takes two int and returns an int
int (Math*)(int, int);

Then you can assign the function pointer like this:

Math = Add;
int result = Math(1,2); // this really calls Add(1,2)
// result will be 3

Math = Sub;
int result = Math(1,2); // this really call Sub(1,2)
// result will be -1
```

Cool, huh?

The next two fields `cbClsExtra` and `cbWndExtra` were originally designed to instruct Windows to save some extra space in the Windows Class to hold extra runtime information. However, most people don't use these fields, and simply set them to 0:

```
winclass.cbClsExtra = 0; // extra class info space
winclass.cbWndExtra = 0; // extra window info space
```

The next field is `hInstance`. This is simply the `hinstance` that is passed to the `WinMain()` function on start-up, so just copy it in from `WinMain()`:

```
winclass.hInstance = hinstance; // assign the application instance
```

The remaining fields relate to graphical aspects of the Window Class, but before I discuss them, I want to take a quick moment to review handles.

You're going to see handles again and again in Windows programs and types. Handles to bitmaps, handles to cursors, handles to everything. Remember, *handles* are just identifiers based on an internal Windows type. In fact, they are really integers, but Microsoft might change this, so it's a good idea to be safe and use the MS types. In any case, you are going to see more and more "handles to blank," so don't trip out on me! And remember: Any `type` prefixed by `h` is usually a handle. Okay, back to the white board.

Playrix Ex. 1023, page 106

04 0672318350 CH02 05•02•2003 2:33 PM Page 82

The next field sets the type of icon that will represent your application. You have the power to load your own custom icon, but for now we are going to use a system icon—which, as you guessed it—you need a handle for. To retrieve a handle to a common system icon, you can use the `LoadIcon()` function:

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

This code loads the standard application icon; boring, but simple. If you're interested in the `LoadIcon()` function, take a look at the Win32 API—there are a number of premade icon values that you can use.

Alright, we're about half way through all the fields. Take another breath and let's forge on to the next field, `hCursor`. This is similar to `hIcon` in that it's a handle to a graphics object. However, it differs in that `hCursor` is the handle to the cursor that will be displayed that represents the mouse pointer when the pointer enters the client region of the window. `LoadCursor()` is used to obtain a handle to a cursor that's a resource or a predefined system cursor. Like I said, we will cover resources a bit later, but they are simply data-like bitmaps, cursors, icons, sounds, and so on that are compiled into your application and can be accessed at runtime. Anyway, here's how we set the cursor to the standard arrow for the Windows Class:

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Again, if you're interested in more cursor styles, check out the Win32 API.

Now we're cooking! We're almost done. The remaining fields are a little more interesting. Let's move on to `hbrBackground`.

Whenever a window is drawn or refreshed, at the very least Windows will repaint the background of the window's client area for you with a predefined color, or *brush* in Window-speak. Hence, `hbrbackground` is a handle to the brush that you want to refresh the window. Brushes, pens, colors, and graphics are all part of the *Graphics Device Interface (GDI)*, which we don't need to worry about because we are primarily going to use DirectX. For now, I'm going to show you how to request a basic system brush with which to paint the window. This is accomplished with the `GetStockObject()` function, as shown in the following line of code:

```
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

CAUTION

Note the cast to `(HBRUSH)`. This is necessary for Visual C++ 6.0+ compilers! However, it won't hurt to add it with other compilers.

`GetStockObject()` is a general function that retrieves a handle to one of Windows' "stock" brushes, pens, palettes, or fonts. `GetStockObject()` takes a single parameter indicating

Playrix Ex. 1023, page 107

04 0672318350 CH02 05•02•2003 2:33 PM Page 83

which one of these resources to load. Table 2.6 contains a list of possible stock objects for brushes and pens only.

TABLE 2.6 Stock Object Identifiers for `GetStockObject()`

Value	Meaning
`BLACK_BRUSH`	Black brush
`WHITE_BRUSH`	White brush
`GRAY_BRUSH`	Gray brush
`LTGRAY_BRUSH`	Light gray brush
`DKGRAY_BRUSH`	Dark gray brush
`HOLLOW_BRUSH`	Hollow brush
`NULL_BRUSH`	Null brush
`BLACK_PEN`	Black pen
`WHITE_PEN`	White pen
`NULL_PEN`	Null pen

In most cases, the background brush of the window will be irrelevant, because after DirectX takes over, it won't matter. It's still good to know.

The next field in the `WNDCLASS` structure is the menu field `lpszMenuName`. This is a `NULL`-terminated ASCII string of the name of the menu resource to load and attach to the window. Because we aren't going to do much with menus, we'll just set it to `NULL`:

```
winclass.lpszMenuName = NULL; // the name of the menu to attach
```

As I mentioned a while ago, each Window Class represents a different type of window that your application can create. Classes are like templates in a manner of speaking; therefore, Windows needs to track and identify them. Hence, the next field `lpszClassName` is for just that. This field is filled with a `NULL`-terminated ASCII string that contains a text identifier for your class. I personally like using identifiers like `WINCLASS1`, `WINCLASS2`, and so forth. It's up to you, but it's better to keep it simple:

```
winclass.lpszClassName = "WINCLASS1"; // the name of the class itself
```

After this assignment, you can refer to the new Window Class by its class name, such as `WINCLASS1`—kinda cool, huh?

Last but not least is the "small" application icon. This is a new addition to the Window Class `WNDCLASSEX` structure and wasn't available in the older `WNDCLASS`. Basically, this handle points to the icon you want displayed on your window's title bar and on the Windows Desktop taskbar. Usually you would load a custom resource, but for now, let's just use one of the standard Windows icons via `LoadIcon()`:

Playrix Ex. 1023, page 108

04 0672318350 CH02 05•02•2003 2:33 PM Page 84

That's it—now let's take a look at the whole class definition at once:

```
WNDCLASSEX winclass; // this will hold the class we create

// first fill in the window class structure
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style   = CS_DBLCLKS ¦ CS_OWNDC ¦ CS_HREDRAW ¦ CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance  = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor   = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = "WINCLASS1";
winclass.hIconSm   = LoadIcon(NULL, IDI_APPLICATION);
```

And of course, if you want to save some typing, you could initialize the structure inline like this:

```
WNDCLASSEX winclass = {
        winclass.cbSize = sizeof(WNDCLASSEX),
        CS_DBLCLKS ¦ CS_OWNDC ¦ CS_HREDRAW ¦ CS_VREDRAW,
        WindowProc,
        0,
        0,
        hinstance,
        LoadIcon(NULL, IDI_APPLICATION),
        LoadCursor(NULL, IDC_ARROW),
        (HBRUSH)GetStockObject(BLACK_BRUSH),
        NULL,
        "WINCLASS1",
        LoadIcon(NULL, IDI_APPLICATION)};
```

It saves typing!

Registering the Windows Class

Now that the Windows Class is defined and stored in `winclass`, you must let Windows know about the new class by *registering* it. This is accomplished via the function `RegisterClassEx()`, which simply takes a pointer to the new class definition:

```
RegisterClassEx(&winclass);
```

Playrix Ex. 1023, page 109

04 0672318350 CH02 05•02•2003 2:33 PM Page 85

CAUTION

Notice that I'm not yet using the class name (`WINCLASS1` in the case of our example). For `RegisterClassEx()`, you must use the actual structure holding the class, because at the point before the call to the function, Windows does not yet know of the existence of the new class. So, the name `WINCLASS1` is not yet available for reference.

Also, to be complete, there is the old `RegisterClass()` function, which is used to register a class based on the older structure `WNDCLASS`.

After the class is registered, we are free to create a window with it. Let's see how to do that, and then revisit the details of the event handler and main event loop to see what kind of processing needs to be done for a Windows application to work.

Creating the Window

To create a window (or any window-like object, such as a control), you use the `CreateWindow()` or `CreateWindowEx()` functions. The latter is a bit more recent and supports an additional style parameter, so we'll use it. This is where the Windows Class that we took so long to dissect piece by piece comes in. When you create a window, you must supply the text name of the window class—which in our case is `WINCLASS1`. This is what identifies your window class and differentiates it from other classes, along with the built-in types like buttons, text boxes, and so on.

Here's the function prototype for `CreateWindowEx()`:

```
HWND CreateWindowEx(
   DWORD dwExStyle,   // extended window style
   LPCTSTR lpClassName, // pointer to registered class name
   LPCTSTR lpWindowName, // pointer to window name
   DWORD dwStyle,    // window style
   int x,        // horizontal position of window
   int y,        // vertical position of window
   int nWidth,     // window width
   int nHeight,     // window height
   HWND hWndParent,   // handle to parent or owner window
   HMENU hMenu,     // handle to menu, or child-window identifier
   HINSTANCE hInstance, // handle to application instance
   LPVOID lpParam);   // pointer to window-creation data
```

If the function is successful, it will return a handle to the newly created window, or `NULL` otherwise.

Playrix Ex. 1023, page 110

04 0672318350 CH02 05•02•2003 2:33 PM Page 86

Most of the parameters are self-explanatory, but let's cover them anyway:

- `dwExStyle`—The extended styles flag is an advanced feature, and for most cases you'll set it to `NULL`. However, if you're interested in all the possible values, take a look at the Win32 SDK Help—there are a lot of them. About the only one I ever use is `WS_EX_TOPMOST`, which makes the window stay on top.
- `lpClassName`—This is the name of the class you want to create a window based on, such as `WINCLASS1`.
- `lpWindowName`—This is a `NULL`-terminated text string containing the title of the window, for example, `My First Window`.
- `dwStyle`—These are the general window flags that describe what the window looks and behaves like—very important! See Table 2.7 for a list of some of the more popular values used for DirectX applications. Of course, you can logically `OR` these values together to get the various features you want.
- `x,y`—Position of the upper left-hand corner of the window in pixel coordinates. If you don't care, use `CW_USEDEFAULT` and Windows will decide.
- `nWidth`, `nHeight`—Width and height of the window in pixels. If you don't care, use `CW_USEDEFAULT` and Windows will decide.
- `hWndParent`—Handle to the parent window if there is one. Use `NULL` if there isn't a parent; then the Desktop will be the parent.
- `hMenu`—Handle to the menu to attach to the window. More on this in the next chapter. Use `NULL` for now.
- `hInstance`—The instance of the application. Use `hinstance` from `WinMain()` here.
- `lpParam`—Advanced. Set to `NULL`.

TABLE 2.7 General Style Values for `dwStyle`

Style	Meaning
`WS_POPUP`	Creates a pop-up window.
`WS_OVERLAPPED`	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the `WS_TILED` style.
`WS_OVERLAPPEDWINDOW`	Creates an overlapped window with the `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MINIMIZEBOX`, and `WS_MAXIMIZEBOX` styles.
`WS_VISIBLE`	Creates a window that is initially visible.

And here's how you would create a basic overlapped window with the standard controls at position (0,0) with a size of (400,400) pixels:

Playrix Ex. 1023, page 111

04 0672318350 CH02 05•02•2003 2:33 PM Page 87


```
HWND hwnd; // window handle

// create the window, bail if problem
if (!(hwnd = CreateWindowEx(NULL, // extended style
              "WINCLASS1",     // class
          "Your Basic Window", // title
         WS_OVERLAPPEDWINDOW ¦ WS_VISIBLE,
          0,0,   // initial x,y
         400,400, // initial width, height
         NULL,   // handle to parent
         NULL,   // handle to menu
         hinstance,// instance of this application
         NULL)))  // extra creation parms
return(0);
```

After the window has been created, it may or may not be visible. However, in this case, we added the style flag `WS_VISIBLE` that does this automatically. If this flag isn't added, use the following function call to manually display the window:

```
// this shows the window
ShowWindow(hwnd, ncmdshow);
```

Remember the `ncmdshow` parameter of `WinMain()`? This is where it comes in handy. Although we have overridden it by adding `WS_VISIBLE`, you would normally send it as the parameter to `ShowWindow()`. The next thing that you might want to do is force Windows to update your window's contents and generate a `WM_PAINT` message. This is accomplished with a call to `UpdateWindow()`:

```
// this sends a WM_PAINT message to window and makes
// sure the contents are refreshed
UpdateWindow();
```

The Event Handler

I don't know about you, but I'm starting to get the hang of this Windows stuff! It's not that bad—it's like a mystery novel, except the mystery is figuring out what language the novel is written in! Let's tackle the main event handler, or at least take a first look at it. Remember that the event handler is a callback function called by Windows from the main event loop whenever an event occurs that your window must handle. Take a look at Figure 2.3 again to refresh your memory of the general data flow.

This event handler is written by *you*, and it handles as many (or as few) events that you want to take care of. The rest you can pass on to Windows and let it deal with them. Of course, the more events and messages your application handles, the more functionality it will have, so keep that in mind.

Playrix Ex. 1023, page 112

04 0672318350 CH02 05•02•2003 2:33 PM Page 88

Before we get into some code, let's talk about some of the details of the event handler and exactly what it does and how it works. First, for each Windows Class that you create, you can have a separate event handler that I will refer to as Windows Procedure, or simply `WinProc` from now on. The `WinProc` is sent messages from the main event loop as messages are received from the user and/or Windows and placed in the main event queue. That's a mental tongue twister, so I'll say it in another way ….

As the user and Windows do stuff, events and messages are generated that are for your window and/or other applications' windows. All of these messages go into a queue, but the ones for your window are sent to your window's own private queue. Then the main event loop retrieves these messages and sends them to your window's `WinProc` to be processed. This is shown in Figure 2.1.

There are literally hundreds of possible messages and variations, so we aren't going to cover them all, but luckily you only have to handle a very few to get a Windows application up and running.

So, in a nutshell the main event loop feeds the `WinProc` with messages and events, and the `WinProc` does something with them. Hence, not only do we have to worry about the `WinProc`, but also the main event loop. However, we will get to this shortly. For now, let's assume that the `WinProc` is simply going to be fed messages. Now that you know what the `WinProc` does, let's take a look at its prototype:

```
LRESULT CALLBACK WindowProc(
         HWND hwnd, // window handle of sender
         UINT msg, // the message id
         WPARAM wparam, // further defines message
         LPARAM lparam); // further defines message
```

Of course, this is just a prototype for the callback. You can call the function anything you want, because you are only going to assign the function's address as a function pointer to `winclass.lpfnWndProc`:

```
winclass.lpfnWndProc = WindowProc;
```

Remember? Anyway, the parameters are fairly self-explanatory.

- `hwnd`—This is the window handle. This is only important if you have multiple windows open with the same Windows Class. `hwnd` is then the only way you can tell which messages are coming from which window. Figure 2.4 shows this possibility graphically.
- `msg`—This is the actual message ID that the `WinProc` should handle. This ID might be one of dozens of main messages.
- `wparam` and `lparam`—These further qualify or subclass the message sent in the `msg` parameter.
- And finally, the return type and declaration specifier are of interest — `LRESULT CALLBACK`. These keywords are a must, so don't forget them!

Playrix Ex. 1023, page 113

04 0672318350 CH02 05•02•2003 2:33 PM Page 89

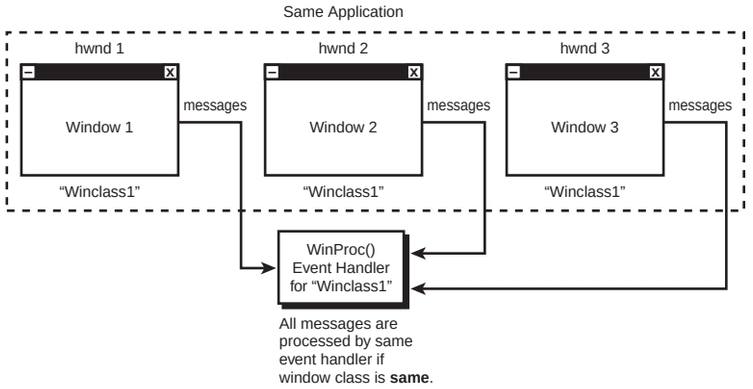


FIGURE 2.4 Multiple windows based on the same class.

What most people do is `switch()` on the `msg` and then write code for each case. Based on `msg` you will know whether you need to further evaluate `wparam` and/or `lparam`. Cool? Let's take a look at some of the possible messages that might come through the `WinProc` and then we'll see a bare bones `WinProc`. Take a look at Table 2.8 to see a short list of some basic message IDs.

TABLE 2.8 A Short List of Message IDs

Value	Meaning
`WM_ACTIVATE`	Sent when a window is activated or becomes the focus.
`WM_CLOSE`	Sent when a window is closed.
`WM_CREATE`	Sent when a window is first created.
`WM_DESTROY`	Sent when a window is about to be destroyed.
`WM_MOVE`	Sent when a window has been moved.
`WM_MOUSEMOVE`	Sent when the mouse has been moved.
`WM_KEYUP`	Sent when a key is released.
`WM_KEYDOWN`	Sent when a key is pressed.
`WM_TIMER`	Sent when a timer event occurs.
`WM_USER`	Enables you to send messages.
`WM_PAINT`	Sent when a window needs repainting.
`WM_QUIT`	Sent when a Windows application is finally terminating.
`WM_SIZE`	Sent when a window has changed size.

(Note: I have highlighted the messages that we are going to make the most use of.)

Take a good look at Table 2.8. Basically, the `WinProc` is sent one or more of these messages as the application runs. The message ID itself will be in `msg`, and any remaining info is stored in `wparam` and `lparam`. Thus, it's always a good idea to reference the online Win32 SDK Help and see what all the parameters of a particular message do.

Playrix Ex. 1023, page 114

04 0672318350 CH02 05•02•2003 2:33 PM Page 90

Fortunately, we are only interested in three messages right now:

- `WM_CREATE`—This message is sent when the window is first created, and gives you a chance to do any setup, initialization, or resource allocation.
- `WM_PAINT`—This message is sent whenever your window's contents need repainting. This can occur for a number of reasons: the window was moved or resized by the user, another application popped up and obscured yours, and so on.
- `WM_DESTROY`—This message is sent to your window when the window is about to be destroyed. Usually, this is a direct result of the user clicking on the window's close icon or closing from the window's system menu. Either way, this is where you should de-allocate all the resources and tell Windows to terminate the application completely by sending a `WM_QUIT` message yourself—more on this later.

So without further ado, let's see a complete `WinProc` that handles all these messages. Take a look at the following listing:

```
LRESULT CALLBACK WindowProc(HWND hwnd,
              UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT  ps;  // used in WM_PAINT
HDC    hdc;  // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
  // do initialization stuff here

    // return success
  return(0);
  } break;

  case WM_PAINT:
  {
  // simply validate the window
  hdc = BeginPaint(hwnd,&ps);
  // you would do all your painting here
    EndPaint(hwnd,&ps);
```

Playrix Ex. 1023, page 115

04 0672318350 CH02 05•02•2003 2:33 PM Page 91


```
    // return success
  return(0);
  } break;

  case WM_DESTROY:
  {
  // kill the application, this sends a WM_QUIT message
  PostQuitMessage(0);

    // return success
  return(0);
  } break;

  default:break;

  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

As you can see, the function is composed of empty space for the most part, which is a good thing! While I'm on the subject of empty space, did you know that most solid matter is composed mostly of empty space? And there's a finite probability that you can walk through a solid wall given enough attempts?

Let's begin with the processing of `WM_CREATE`. Here all the function does is `return(0)`. This simply tells Windows that we handled it, so don't take any more action. Of course, you could do all kinds of initialization in the `WM_CREATE` message, but that's up to you.

The next message `WM_PAINT` is very important. This message is sent whenever your window needs repainting. This means that you have to do the repainting, for the most part. This isn't going to matter for DirectX games because we are going to redraw the screen 30—60 fps (frames per second), but for a normal Windows application, it does. Nevertheless, for our DirectX applications, we don't want Windows to think that we aren't repainting the screen, so we have to let it know we are processing the `WM_PAINT` messages.

To accomplish this feat, you must "validate the client rectangle of the window." There are a number of ways to do this, but the simplest is with a call to `BeginPaint()` and `EndPaint()`. This calling pair validates the window and fills the background with the background brush previously stored in the Windows Class variable `hbrBackground`. Here's the code once again for the validation:

Playrix Ex. 1023, page 116

04 0672318350 CH02 05•02•2003 2:33 PM Page 92


```
    // begin painting
  hdc = BeginPaint(hwnd,&ps);
  // you would do all your painting here
    EndPaint(hwnd,&ps);
```

There are a couple of things going on here that I want to address. First, notice that the first parameter to each call is the window handle `hwnd`. This is necessary because the `BeginPaint()`/`EndPaint()` function can potentially paint in any window of your application; so the window handle indicates which window you're interested in painting in. The second parameter is the address of a `PAINTSTRUCT` structure that contains the rectangle that you must redraw. That is, it's possible that only a small region needs repainting, so this structure defines that region. Here's what a `PAINTSTRUCT` looks like:

```
typedef struct tagPAINTSTRUCT
    {
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
    } PAINTSTRUCT;
```

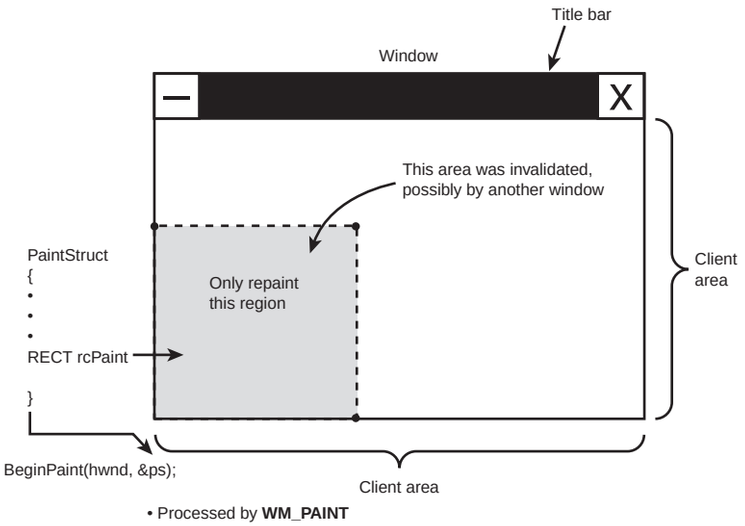


FIGURE 2.5 Repainting the invalid region only.

You mostly don't need to worry about this until later, when we talk about the GDI, but the most important field is `rcPaint`, which is a `RECT` structure that contains the minimum

Playrix Ex. 1023, page 117

04 0672318350 CH02 05•02•2003 2:33 PM Page 93

rectangle that needs to be repainted. Take a look at Figure 2.5 to see this graphically. You see, Windows tries to do the least amount of work possible, so when the contents of a window are mangled, Windows at least tries to tell you the smallest rectangle that you can repaint to restore the contents. If you're interested in the `RECT` structure, it's nothing more than the four corners of a rectangle, as shown in the following:

```
typedef struct tagRECT
    {
    LONG left;  // left x-edge of rect
    LONG top;  // top y-edge of rect
    LONG right; // right x-edge of rect
    LONG bottom; // bottom y-edge of rect
    } RECT;
```

The last thing you'll notice about the call to `BeginPaint()` is that it returns a handle to a graphics context, or `hdc`:

```
HDC hdc; // handle to graphics context
hdc = BeginPaint(hwnd,&ps);
```

A *graphics context* is a data structure that describes the video system and drawing surface. It's magic as far as we are concerned (but it wouldn't hurt you to take a look at its data structure in the header files); you just have to retrieve one if you want to do any graphics. That's about it for the `WM_PAINT` message—for now.

The `WM_DESTROY` message is actually quite interesting. `WM_DESTROY` is sent when the user closes the window. However, this only closes the window, not the application. The application will continue to run, but without a window. Therefore, you need to do something about this. In most cases, when the user kills the main window, he intends for the application to terminate. Thus, you must facilitate this by sending a quit message yourself! The message is called `WM_QUIT`. Because it's so common a message, there is a function to send it for you called `PostQuitMessage()`.

All you need to do in the `WM_DESTROY` handler is clean up everything and then tell Windows to terminate your application with a call to `PostQuitMessage(0)`. This in turn puts a `WM_QUIT` into the message queue, which at some point causes the main event loop to bail.

There are a couple of details you should be aware of in the `WinProc` handler we have been analyzing. First, I'm sure you have noticed the `return(0)` after each handler's body. This serves two purposes: to exit the `WinProc`, and to tell Windows that we have handled the message. The second important detail is the *default message handler* `DefaultWindowProc()`. This function is a *pass through*, which passes messages that you don't process on to Windows for default processing. Therefore, if you don't handle the message, make sure to always end all your event handler functions with a call like this:

Playrix Ex. 1023, page 118

04 0672318350 CH02 05•02•2003 2:33 PM Page 94


```
// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));
```

I know this might all seem like overkill and more trouble than it's worth. Nevertheless, after you have a basic Windows application skeleton, then you just copy it and add your own code. In fact, after we have our virtual computer set up, we can use it and almost forget that any Windows stuff is going on. Anyway, stay with me and let's move on to the last part: the main event loop.

The Main Event Loop

The hard part is over! The main event loop is so simple that I'm just going to blurt it out and then talk about it, so here it is:

```
// enter main event loop
while(GetMessage(&msg,NULL,0,0))
   {
   // translate any accelerator keys
   TranslateMessage(&msg);

   // send the message to the window proc
   DispatchMessage(&msg);
   } // end while
```

That's it? Yup! Let's see what's going on here, shall we? The main `while()` executes as long as `GetMessage()` returns a nonzero value. `GetMessage()` is the workhorse of the main event loop, and its sole purpose is to get the next message from the event queue and process it. You'll notice that there are four parameters to `GetMessage()`. The first one is important to us, but the remaining are set to `NULL` and zeros. Here's the prototype for reference:

```
BOOL GetMessage(
   LPMSG lpMsg,     // address of structure with message
   HWND hWnd,      // handle of window
   UINT wMsgFilterMin, // first message
   UINT wMsgFilterMax); // last message
```

The `msg` parameter is (you guessed it) the storage where Windows will place the next message. However, unlike the `msg` parameter for `WinProc()`, this `msg` is a complex data structure, rather than just an integer ID. Remember, by the time a message gets to the `WinProc`, it has been "cooked" and split into its constituent parts. Anyway, here is the `MSG` structure:

```
typedef struct tagMSG
    {
    HWND hwnd;   // window where message occurred
```

Playrix Ex. 1023, page 119

04 0672318350 CH02 05•02•2003 2:33 PM Page 95


```
UINT message; // message id itself
WPARAM wParam; // sub qualifies message
LPARAM lParam; // sub qualifies message
DWORD time;  // time of message event
POINT pt;   // position of mouse
} MSG;
```


Starting to make sense? Notice that the parameters to `WinProc()` are all contained within this structure, along with some others, like the time and position of the mouse when then event occurred.

We know `GetMessage()` retrieves the next message from the event queue, but then what? `TranslateMessage()` is called next. `TranslateMessage()` is a virtual accelerator key translator, or in other words, an input cooker. Just call it and don't worry about what it does. The final function `DispatchMessage()` is where all the action occurs. After the message is retrieved with `GetMessage()` and potentially processed and translated a bit with `TranslateMessage()`, the actual `WinProc()` is called by the call to `DispatchMessage()`.

`DispatchMessage()` makes the call to the `WinProc`, sending the appropriate parameters from the original `MSG` structure. Figure 2.6 shows the whole process in its final glory.

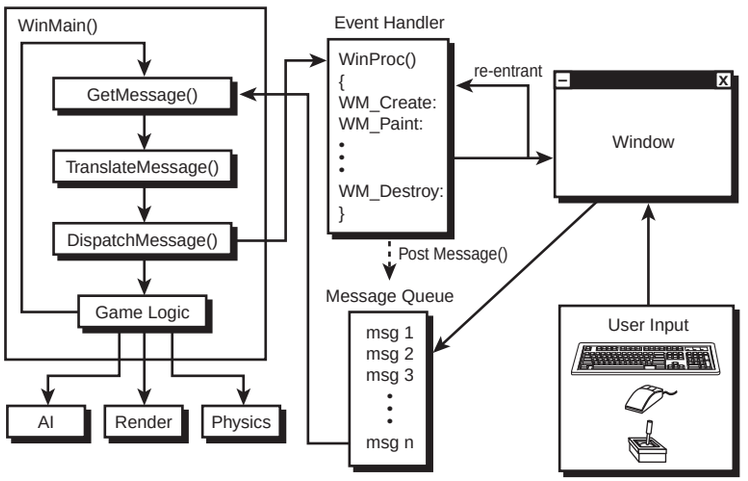


FIGURE 2.6 The mechanics of event loop message processing.

That's it—you're a Windows expert! The rest is just details, but if you grasp the concepts we just covered and the importance of the event loop, event handler, and so on, that's 90% of the battle. With that in mind, take a look at the following listing—it's a complete Windows program that creates a single window and waits for you to close it.

Playrix Ex. 1023, page 120

04 0672318350 CH02 05•02•2003 2:33 PM Page 96


```
// DEMOII2_3.CPP - A complete windows program

// INCLUDES ///////////////////////////////////////////////
#define WIN32_LEAN_AND_MEAN // just say no to MFC

#include <windows.h>  // include all the windows headers
#include <windowsx.h> // include useful macros
#include <stdio.h>
#include <math.h>

// DEFINES ////////////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WINCLASS1"

// GLOBALS ////////////////////////////////////////////////


// FUNCTIONS //////////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
         UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT  ps;  // used in WM_PAINT
HDC    hdc;  // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
  // do initialization stuff here

    // return success
  return(0);
  } break;

  case WM_PAINT:
  {
  // simply validate the window
  hdc = BeginPaint(hwnd,&ps);
```

Playrix Ex. 1023, page 121

04 0672318350 CH02 05•02•2003 2:33 PM Page 97


```
  // you would do all your painting here
    EndPaint(hwnd,&ps);

    // return success
  return(0);
  } break;

  case WM_DESTROY:
  {
  // kill the application, this sends a WM_QUIT message
  PostQuitMessage(0);

    // return success
  return(0);
  } break;

  default:break;

  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////
int WINAPI WinMain(HINSTANCE hinstance,
      HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow)
{

WNDCLASSEX winclass; // this will hold the class we create
HWND    hwnd;   // generic window handle
MSG    msg;     // generic message

// first fill in the window class stucture
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style   = CS_DBLCLKS ¦ CS_OWNDC ¦
         CS_HREDRAW ¦ CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra  = 0;
winclass.cbWndExtra  = 0;
```

Playrix Ex. 1023, page 122

04 0672318350 CH02 05•02•2003 2:33 PM Page 98


```
winclass.hInstance  = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;
winclass.hIconSm   = LoadIcon(NULL, IDI_APPLICATION);

// register the window class
if (!RegisterClassEx(&winclass))
  return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL, // extended style
              WINDOW_CLASS_NAME,  // class
        "Your Basic Window", // title
        WS_OVERLAPPEDWINDOW ¦ WS_VISIBLE,
         0,0,    // initial x,y
        400,400,    // initial width, height
        NULL,    // handle to parent
        NULL,    // handle to menu
        hinstance,   // instance of this application
        NULL)))    // extra creation parms
return(0);

// enter main event loop
while(GetMessage(&msg,NULL,0,0))
   {
   // translate any accelerator keys
   TranslateMessage(&msg);

   // send the message to the window proc
   DispatchMessage(&msg);
   } // end while

// return to Windows like this
return(msg.wParam);

} // end WinMain

////////////////////////////////////////////////////////////
```

Playrix Ex. 1023, page 123

04 0672318350 CH02 05•02•2003 2:33 PM Page 99

To compile DEMOII2_3.CPP, simply create a Win32 .EXE application and add DEMOII2_3.CPP to the project. Or if you like, you can run the precompiled program DEMOII2_3.EXE from the CD-ROM. Figure 2.7 shows the program in action. It's not much to look at, but what do you want—this is a paperback book!



FIGURE 2.7 Screenshot of DEMOII2_3.EXE in action.

There are a couple of issues that I want to hit you with before moving on. First, if you take a closer look at the event loop, it doesn't look like it works in real time—while the program waits for a message via GetMessage(), the main event loop is basically blocked. This is very true, and we must get around it somehow, because we need to perform our game processing continuously and handle Windows events if and when they come.

Making a Real-Time Event Loop

A real-time non-waiting/blocking event loop is easy to make. All we need is a way to test whether there is a message in the message queue. If there is, we can process it; otherwise, we can continue processing other game logic and repeat. The function that performs just this test is called PeekMessage(). Its prototype is almost identical to GetMessage()'s, as shown here:

```
BOOL PeekMessage(
   LPMSG lpMsg,    // pointer to structure for message
   HWND hWnd,     // handle to window
   UINT wMsgFilterMin, // first message
   UINT wMsgFilterMax, // last message
   UINT wRemoveMsg);  // removal flags
```

Returns nonzero if a message is available.

Playrix Ex. 1023, page 124

04 0672318350 CH02 05•02•2003 2:33 PM Page 100

The difference is in the last parameter. This controls how the messages should be retrieved from the message queue. The valid flags for `wRemoveMsg` are the following:

- `PM_NOREMOVE`—Messages are not removed from the queue after processing by `PeekMessage()`.
- `PM_REMOVE`—Messages are removed from the queue after processing by `PeekMessage()`.

Taking these two possibilities into consideration, we can do one of two things: use `PeekMessage()` with `PM_NOREMOVE`, and if there is a message, call `GetMessage()`; or use `PM_REMOVE` and use `PeekMessage()` itself to retrieve messages. We'll use the latter. Here's the core logic, changed to reflect this new technique in the main event loop:

```
while(TRUE)
  {
  // test if there is a message in queue, if so get it
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // test if this is a quit
     if (msg.message == WM_QUIT)
       break;

    // translate any accelerator keys
    TranslateMessage(&msg);

    // send the message to the window proc
    DispatchMessage(&msg);
    } // end if

    // main game processing goes here
  Game_Main();
  } // end while
```

I've highlighted important points in the code. The first section highlighted is shown here:

```
     if (msg.message == WM_QUIT)
       break;
```

This is how we must detect to bail out of the infinite `while(TRUE)` loop. Remember, when a `WM_DESTROY` message is processed in the `WinProc`, it's our job to send a `WM_QUIT` message via the call to `PostQuitMessage()`. The `WM_QUIT` then trickles through the event queue, and we can detect it to bail out of the loop.

The last section of highlighted code simply indicates where you would put the call to your main game code loop. But remember, the call to `Game_Main()` (or whatever you call it)

Playrix Ex. 1023, page 125

04 0672318350 CH02 05•02•2003 2:33 PM Page 101

must return after one frame of animation or game logic; otherwise, messages won't be processed by the main windows event loop.

As an example of this new real-time structure that is more appropriate for game logic processing, take a look at the source `DEMOII2_4.CPP` and the associated `DEMOII2_4.EXE` on the CD-ROM. This structure will in fact be our model for the virtual computer used for the remainder of the book. We will build on it in the next chapter.

Now that you have a handle on basic Windows programming, let's talk about the graphics, sound, and input interface that we'll be using for our games: DirectX.

DirectX and COM Crash Course

DirectX might take one more bit of control from you as a programmer, but in truth it's worth its weight in Monolithic DIMMS (Dual Inline Memory Modules). DirectX is basically a system of software that abstracts graphics, audio, input, networking, installation, and more, so no matter what the hardware configuration of a particular PC, you can use the same code. In addition, DirectX technology is many times faster and more robust than GDI and/or MCI (the Media Control Interface), which is native to Windows. Figure 2.8 illustrates how you would make a Windows game with and without DirectX. Notice how clean and elegant the DirectX solution is.

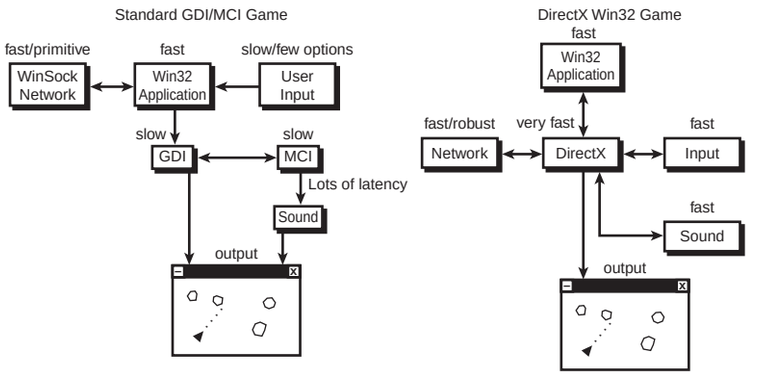


FIGURE 2.8 DirectX versus GDI/MCI.

So how does DirectX work? DirectX gives you almost hardware-level control of all devices. This is possible through a technology called *COM (Component Object Model)* and sets of drivers and libraries written by both Microsoft and the hardware vendors themselves. Microsoft came up with a set of conventions, functions, variables, data structures, and so on that must be used by the hardware vendors when implementing drivers to talk to the hardware.

Playrix Ex. 1023, page 126

04 0672318350 CH02 05•02•2003 2:33 PM Page 102

As long as these conventions are followed, you don't need to worry about the details of the hardware—you just make calls to DirectX and DirectX handles the details for you. No matter what kind of video card, sound card, input device, network card, or whatever, as long as there is DirectX support, your program will be able to use it without you knowing anything about it!

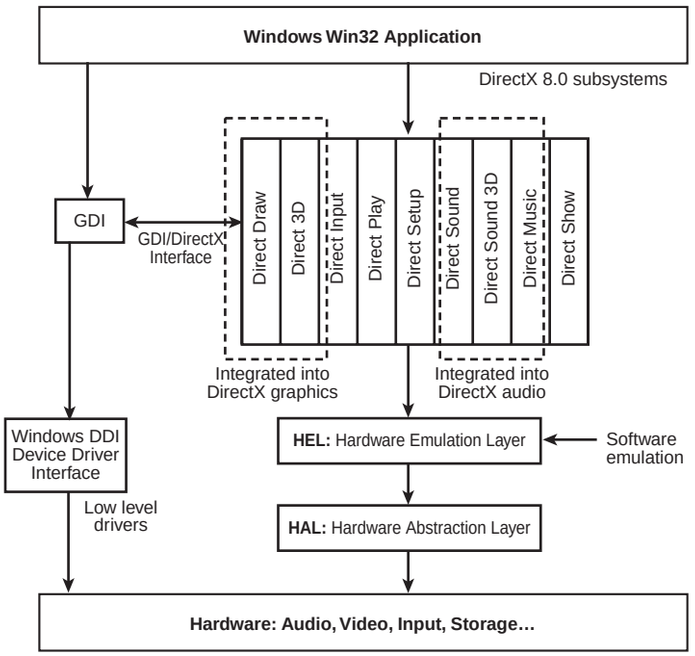


FIGURE 2.9 The architecture of DirectX 8.0+ and its relationship to Win32.

Currently there are a number of DirectX components that make up the DirectX 8.0+ foundation system. They are listed here and shown graphically in Figure 2.9.

- DirectDraw
- DirectSound
- DirectSound3D
- DirectMusic
- DirectInput
- DirectPlay
- DirectSetup

Playrix Ex. 1023, page 127

04 0672318350 CH02 05•02•2003 2:33 PM Page 103

- Direct3DRM
- Direct3DIM

In DirectX 8.0, DirectDraw was merged with Direct3D and renamed DirectGraphics, and DirectSound and DirectMusic were merged and renamed DirectAudio. Thus, under DirectX 8.0+, there is no more DirectDraw. However, one of the rules of DirectX and COM is that you can always request a previous interface. Thus, in this book we will use DirectDraw under DirectX 7.0a for graphics, and the newer 8.0+ interfaces for sound and input. In any case, we are going to abstract graphics, sound, and input into our virtual computer, so truthfully even DirectX 3.0 would work! We will use the latest versions of DirectX (except for DirectDraw) to be as up-to-date as possible.

The HEL and HAL

You might notice on Figure 2.9 that there are two layers under DirectX called the *HEL (Hardware Emulation Layer)* and the *HAL (Hardware Abstraction Layer)*. Here's the deal—DirectX is a very forward-looking design, so it assumes advanced features are implemented by hardware. However, if the hardware doesn't support some feature, what happens? This is the basis of the dual-mode HAL and HEL design.

- HAL—The Hardware Abstraction Layer is the "to-the-metal layer." It talks directly to the hardware. This layer is usually the device driver from the vendor, and you communicate to it directly through generic DirectX calls. HAL is used when the feature you're requesting is supported directly by the hardware, and is thus accelerated.
- HEL—The Hardware Emulation Layer is used when the hardware doesn't support the feature that you are requesting. Let's suppose that you ask the video card to rotate a bitmap. If the hardware doesn't support rotation, the HEL kicks in and software algorithms take over. Obviously, this is slower, but the point is that it does NOT break your program. It will still work—just slower. In addition, the switching between the HAL and HEL is transparent to you. If you request DirectX to do something and the HAL does it directly, hardware will do it; otherwise, a software emulation will be called to get the job done with HEL.

You might be thinking that there are a lot of layers of software here. That's a concern, but the truth is that DirectX is so clean that the only penalty you take for using it is maybe an extra function call or two. A small price to pay for 2D/3D graphics, network, and audio acceleration. Can you imagine writing drivers to control all the video accelerators on the market? Trust me, it would take literally thousands of man-years—it just can't be done. DirectX is really a massively distributed engineering effort by Microsoft and all the hardware vendors to bring you an ultra-high performance standard.

Playrix Ex. 1023, page 128

04 0672318350 CH02 05•02•2003 2:33 PM Page 104

The DirectX Foundation Classes in Depth

Now let's take a quick look at each of the DirectX components and what they do:

- DirectDraw (DirectX 7.0a only, not available in DirectX 8.0+)—This is the primary rendering and 2D bitmap engine that controls the video display. It is the conduit that all graphics must go through. This is probably the most important of all the DirectX components. The DirectDraw object more or less represents the video card(s) in your system.
- DirectSound—This is the sound component of DirectX. However, it only supports digital sound—no MIDI. However, this component makes your life a hundred times easier, because no longer do you have to license a third-party sound system to do your sound. Sound programming is a black art, and in the past no one wanted to keep up with writing all the drivers for all the sound cards. Hence, a couple of vendors cornered the market on sound libraries: Miles Sound System and DiamondWare Sound Toolkit. Both were very capable systems that enabled you to simply load and play digital and MIDI sounds from your DOS or Win32 programs. However, with the introduction of DirectSound, DirectSound3D, and the latest DirectMusic components, third-party libraries are much less useful now.
- DirectSound3D—This is the 3D sound component of DirectSound. It enables you to position 3D sounds in space and simulate 3D sound as if objects are floating around the room! This technology is relatively new, but maturing quickly. Today most sound cards support hardware-accelerated 3D effects, including Doppler shift, refraction, reflection, and more. However, if software emulation is used, all of this stuff comes to a halt!
- DirectMusic—DirectMusic is the missing MIDI technology that DirectSound didn't support. But more than that, DirectMusic has a new *DLS* system *(Downloadable Sounds)* that enables you to create digital representations of instruments and then play them back with MIDI control. This is much like a Wave Table synthesizer, but in software. Also, DirectMusic has a new Performance Engine that is an artificial intelligence system of sorts, and can make changes to your music in real time based on templates you give it. In essence, the system can create new music on the fly. Wild, huh?
- DirectAudio—In version 8.0+ of DirectX, DirectSound and DirectMusic were merged into a unified interface. You can still use them separately if you like, but if you prefer you can now use the new DirectAudio interface as well.
- DirectInput—This system handles all input devices, including the mouse, keyboard, joystick, paddles, track balls, and so forth. Moreover, DirectInput now supports force feedback devices. These are devices that have electro-mechanical actuators and force sensors, enabling you to physically manifest forces that the user can feel. It's going to really put the cybersex industry into overdrive!

Playrix Ex. 1023, page 129

04 0672318350 CH02 05•02•2003 2:33 PM Page 105

- DirectPlay—This is the networking aspect of DirectX. It enables you to make abstract "connections" using the Internet, modems, direct connection, or any other kind of medium that might come up. The cool thing about DirectPlay is that it enables you to make these connections without knowing anything about networking. You don't have to write drivers, use sockets, or anything like that. In addition, DirectPlay supports the concepts of *sessions*, which are games in progress, and *lobbies*, which are places for gamers to congregate to play. Also, DirectPlay doesn't force you into any kind of multiplayer network architecture—it's up to you how you want to do things. All DirectPlay does is send and receive packets for you. What they contain and if they are reliable is up to you.
- Direct3DRM (DirectX 7.0a only, not available in DirectX 8.0+)—This is Direct3D Retained Mode, which is a high-level object and frame-based 3D system that you can use to create basic 3D programs. It takes advantage of 3D acceleration, but isn't the fastest thing in the world. It's great for making walkthru programs, 3D model manipulators, or extremely slow demos.
- Direct3DIM—This is Direct3D Immediate Mode, which is the low-level 3D support for DirectX. Originally, this was incredibly hard to work with, and was the cause for many flame wars with OpenGL. The old Immediate Mode used what are called *execute buffers*—basically, arrays of data and instructions that you would create that describe the scene to be drawn—very ugly. However, since DirectX 5.0, Immediate Mode now supports a much more OpenGL-like interface through the `DrawPrimitive()` and `DrawIndexedPrimitive()` functions. This enables you to send triangle strips, fans, and so on to the rendering engine, and make state changes with function calls rather than execute buffers. Although this book is a software-based 3D game book, Direct3D Immediate Mode will be covered at the end of the book for completion's sake.
- DirectGraphics—In DirectX 8.0+, DirectDraw is no longer supported, but that's not entirely true—from a philosophical point of view, at least. What Microsoft did was merge all the graphics software into a single interface called DirectGraphics that handles everything. It doesn't have any DirectDraw interfaces, but you can perform 2D with Direct3D interfaces if you want, or just use IDirectDraw7 and standard DirectDraw—I know, it's a mess, huh?
- DirectSetup/AutoPlay—These are quasi-DirectX components that enable you to install DirectX from your application on the user's machine and start the game up directly when the CD is placed in the system. DirectSetup is a small set of functions that basically loads the runtime DirectX files on a user's machine and registers them in the registry. AutoPlay is the standard CD subsystem that looks for the `AUTOPLAY.INF` file on the CD root, and if found, executes the batch command functions in the file.

Playrix Ex. 1023, page 130

04 0672318350 CH02 05•02•2003 2:33 PM Page 106

Finally, you might be wondering what the deal is with all the versions of DirectX—it seems to be revised on a six-month basis. This is for the most part true. It's a manifestation of the business we are in—graphics and game technology move very fast. However, because DirectX is based on COM technology, programs that you write for, say, DirectX version 3.0, are guaranteed to work on DirectX version 8.0, 9.0, and so on because of their COM implementation. Let's see how that works ….

Flash Introduction to COM

COM was originally invented many years back as a simple white paper on a new software paradigm that was similar to how computer chips or LEGO blocks work—you simply plug them together and they work. Both computer chips and LEGO blocks know how to be computer chips and LEGO blocks, so everything works out automatically. To implement this kind of technology with software, you need some kind of very generic interface that can take on the form of any type of function set you can imagine. This is what COM does.

One of the cool things about computer chips is that when you add more computer chips to a design, you don't have to tell all the other chips that you changed something. However, as you know, this is a little harder with software programs. You at least have to recompile to make an executable. Fixing this problem is another goal of COM. You should be able to add new features to a COM object without breaking software that uses the old COM object. In addition, COM objects can be changed without recompiling the original program, which is very cool.

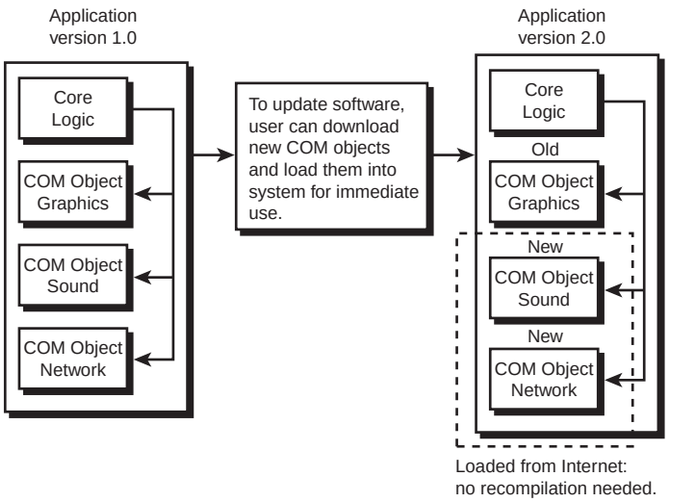


FIGURE 2.10 An overview of COM.

Playrix Ex. 1023, page 131

04 0672318350 CH02 05•02•2003 2:33 PM Page 107

Because you can upgrade COM objects without recompiling your program, you can also upgrade your software without patches and new versions. For example, suppose that you have a program that uses three COM objects: one that implements graphics, one for sound, and one for networking, as shown in Figure 2.10. Now imagine that you sell 100,000 copies of your program that uses these COM objects, but you don't want to send out 100,000 upgrades! To update the graphics COM object, all you do is give users the new COM object for graphics, and the program will automatically use it. No recompiling, linking, or anything like that—easy. Of course, all this technology is very complex at the low level, and writing your own COM objects is a bit challenging, but using them is easy.

The next question is how are COM objects distributed or contained, given their plug-n-play nature? The answer is that there are no rules on this. However, in most cases COM objects are *.DLLs (Dynamic Link Libraries),* which can be downloaded or supplied with the program that uses them. This way they can be easily upgraded and changed. The only problem with this is that the program that uses the COM object must know how to load the COM object from a .DLL—but we'll get to that later.

What Exactly Is a COM Object?

A COM object is really a C++ class or a set of C++ classes that implement a number of *interfaces*. These interfaces are used to communicate to the COM object. Basically, an interface is a set of functions that perform a service. Take a look at Figure 2.11. Here we see a single COM object that has three interfaces named `IGRAPHICS`, `ISOUND`, and `IINPUT`.

Each one of these interfaces has a number of functions that you can call (when you know how) to do work. So, a single COM object can have one or more interfaces, and you may have one or more COM objects. Moreover, the COM specification states that all interfaces you create must be derived from a special base class interface called `IUnknown`. For you C programmers, all this means is that `IUnknown` is like the starting point to build the interface from.

Let's take a look at the `IUnknown` class definition:

```
struct Iunknown
{

// this function is used to retrieve other interfaces
virtual HRESULT __stdcall QueryInterface(const IID &iid, (void **)ip) = 0;

// this is used to increment interfaces reference count
virtual ULONG __stdcall AddRef() = 0;

// this is used to decrement interfaces reference count
virtual ULONG __stdcall Release() = 0;

};
```

Playrix Ex. 1023, page 132

04 0672318350 CH02 05•02•2003 2:33 PM Page 108

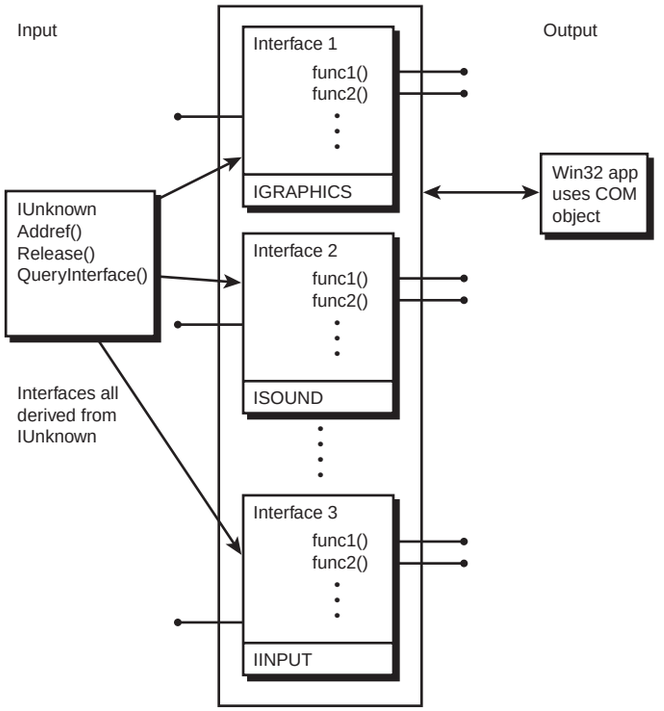


FIGURE 2.11 The interfaces of a COM object.

NOTE

Notice that all methods are pure and virtual. In addition, the methods use `__stdcall` in deference to the standard C/C++ calling convention. Remember that `__stdcall` pushes the parameters on the stack from right to left.

Even if you're a C++ programmer, the preceding class definition might look a bit bizarre if you're rusty on virtual functions. Anyway, let's dissect `IUnknown` and see what's up. All interfaces derived from `IUnknown` must implement at very minimum each of the methods `QueryInterface()`, `AddRef()`, and `Release()`.

`QueryInterface()` is the key to COM—it is used to request a pointer to the interface functions that you desire. To make the request happen, you must have an interface ID. This interface ID is a unique number 128 bits long that you assign to your interface. There are 2^{128} different possible interface IDs, and I guarantee that in a billion years, even if everybody on this planet does nothing but make COM objects, we won't run out! More on the ID when we get to a real example a little later in the chapter. Furthermore, one of the rules

Playrix Ex. 1023, page 133

04 0672318350 CH02 05•02•2003 2:33 PM Page 109

of COM is that if you have an interface, you can always request any other interface from it as long as it's from the same COM object. This means that you can get anywhere from anywhere else. Take a look at Figure 2.12 to see this graphically.

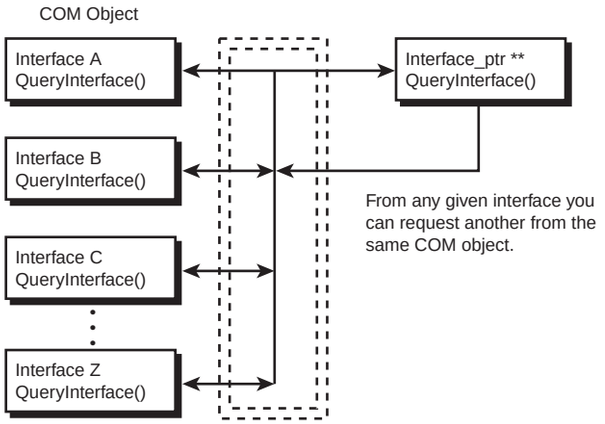


FIGURE 2.12 Navigating the interfaces of a COM object.

`AddRef()` is a curious function. COM objects use a technique called *reference counting* to track their life. This is because one of the specifications of COM is that it's not language-specific. Hence, `AddRef()` is called when a COM object is created and when interfaces are created to track how many references there are to the objects. If COM objects were to use `malloc()` or `new[]`, that would be C/C++-specific. When the reference count drops to `0`, the object(s) are destroyed internally.

TIP

Usually you don't have to call `AddRef()` yourself on interfaces or COM objects; it's done internally by the `QueryInterface()` function. However, sometimes you might have to if you want to up the reference count to trick the COM object into thinking that there are more references to it than there really is.

This brings us to a problem—if COM objects are C++ classes, how can they be created or used in Visual Basic, Java, Active X, and so on? It just so happens that the designers of COM used virtual C++ classes to implement COM, but you don't need to use C++ to access them, or even to create them. As long as you create the same binary image that a Microsoft C++ compiler would when creating a virtual C++ class, the COM object will be COM-compliant. Of course, most compiler products have extras or tools to help make COM objects, so that's not too much of a problem. The cool thing about this is that you can write a COM object in C++, Visual Basic, or Delphi, and then that COM object can be used by any of those languages! A binary image in memory is a binary image in memory.

Playrix Ex. 1023, page 134

04 0672318350 CH02 05•02•2003 2:33 PM Page 110

`Release()` is used to decrement the reference count of a COM object or interface. In most cases, you must call this function yourself when you're done with an interface. However, sometimes if you create an object and then from that object create another object, killing the calling `Release()` on the parent will trickle down and `Release()` the child or derived object, but either way, it's a good idea to `Release()` in the opposite order that you queried.

Creating and Using DirectX COM Interfaces

At this point, I think we all understand that COM objects are collections of interfaces that are simply function pointers, and are more specifically, `VTABLE`s. Hence, all you need to do to work with a DirectX COM object is create it, retrieve an interface pointer, and then make calls to the interface using the proper syntax. Of course, this is all for your edification, because we are going to use an API that builds on all this stuff. I doubt you will ever make a COM call (maybe for DirectMusic and DirectInput), but it's good to know. As an example, I'll use the main DirectDraw interface to show how this is done.

First off, to experiment with DirectDraw you need three things:

- The DirectDraw runtime COM object(s) and .DLLs must be loaded and registered. This is what the DirectX installer does.
- You must include the `DDRAW.LIB` import library in your Win32 programs, so that the wrapper functions you call are linked in.
- You need to include `DDRAW.H` in your program, so the compiler can "see" the header information, prototypes, and data types for DirectDraw.

With that in mind, here's the data type for a DirectDraw 1.0 interface pointer:

```
LPDIRECTDRAW lpdd = NULL;
```

To create a DirectDraw 1.0 COM object and retrieve an interface pointer to the DirectDraw object (which represents the video card), all you need to do is use the wrapper function `DirectDrawCreate()`:

```
DirectDrawCreate(NULL, &lpdd, NULL);
```

Don't worry about the parameters. If you're interested, take a look at the DirectX SDK, but 99 out of a 100 times, the function will look as it does in the preceding. That is, only one parameter is used, and that's the address of the interface pointer to fill in with the DirectDraw COM interface. So for now, just believe that this call creates a DirectDraw object and assigns the interface pointer to `lpdd`.

NOTE

Of course, there is a lot going on in the function. It opens a .DLL, loads it, makes calls and does about a million other things, but you don't have to worry about it.

Playrix Ex. 1023, page 135

04 0672318350 CH02 05•02•2003 2:33 PM Page 111

Now we're in business and can make calls to DirectDraw. But wait a minute! We don't know the methods or functions that are available—that's why you're reading this book <BG>. However, as an example, here's how you would set the video mode to 640×480 with 8-bit (256) color:

```
lpdd->SetVideoMode(640, 480, 8);
```


Is that simple or what? About the only extra work done is the pointer de-reference from the DirectDraw interface pointer `lpdd`—that's it. Of course, what's really happening is a lookup in the virtual table of the interface, but we aren't concerned about that.

In essence, any call to DirectX takes the form

```
interface_pointer->method_name(parameter list);
```

Also, for any other interfaces that you might want to work with (for example, Direct3D), you will get from the original DirectDraw interface by using `QueryInterface()`. Because there are multiple versions of DirectX floating around, Microsoft stopped writing wrapper functions a while ago to retrieve the latest interface. What this means is that you must manually retrieve the latest DirectX interface yourself with `QueryInterface()`. Let's take a look at that.

Querying for Interfaces

The weird thing about DirectX is that all the version numbers are out of sync. This is a bit of a problem, and definitely a cause for confusion. When the first version of DirectX came out, DirectDraw interface was named like this:

```
IDIRECTDRAW
```

Then when DirectX 2.0 came out, DirectDraw was upgraded to version 2.0, so we had

```
IDIRECTDRAW

IDIRECTDRAW2
```

Now at version 7.0a, we have this:

```
IDIRECTDRAW

IDIRECTDRAW2

IDIRECTDRAW4

IDIRECTDRAW7
```

Playrix Ex. 1023, page 136

04 0672318350 CH02 05•02•2003 2:33 PM Page 112

And with version 8.0 and 9.0, there is no more `IDIRECTDRAW` upgrade because it was merged with Direct3D, but the other interfaces are still there:

```
IDIRECTDRAW

IDIRECTDRAW2

IDIRECTDRAW4

IDIRECTDRAW7
```

I mention 8.0 and 9.0 in the same breath because they are almost identical—there were no major changes, as there were from 7.0 to 8.0. The only differences in 9.0 are more advanced 3D support for shader programming, some more DirectPlay software, and a number of other enhancements. But more or less the major changes happened in the transition from 7.0 to 8.0—things will never be the same. Now you might be thinking "Wait a minute—what happened to interfaces 3 and 5?" I have no idea, and this is the problem. The idea is, even though you are using DirectX 8.0+, it doesn't mean that the interfaces are up to that version. Moreover, they can all be out of sync. DirectX 6.0 might have DirectDraw interfaces up to `IDIRECTDRAW4`, but DirectSound is only up to 1.0, which is simply called `IDIRECTSOUND`. You can see the mess we are in! The moral of the story is that whenever you use a DirectX interface, make sure that you are using the latest version. If you're not sure, use the revision 1 interface pointer from the generic "create" function to get the latest version. Here's an example of what I'm talking about.

`DirectDrawCreate()` returns a revision 1.0 interface pointer, but DirectDraw is really up to `IDIRECTDRAW7`, so how do we take advantage of this new functionality?

TIP

If you're having a panic attack over this stuff, don't feel alone. I was using the version 1.0 interfaces up until version 5.0 of DirectX because the DirectX docs are nebulous in this area—typical!

There are two ways to do this from the DirectDraw 1.0 interface: With low-level COM functions, or with `QueryInterface()`; let's use the latter. The process goes like this:

1. You create the DirectDraw COM interface with a call to `DirectDrawCreate()`. This returns a boring `IDIRECTDRAW` interface pointer.
2. Using the pointer retrieved from step 1, you make a call to `QueryInterface()` using the interface ID (GUID) for `IDIRECTDRAW7`, and you retrieve the new interface.

Playrix Ex. 1023, page 137

04 0672318350 CH02 05•02•2003 2:33 PM Page 113

Here's an example of querying for the DirectDraw7 interface:

```
LPDIRECTDRAW lpdd;  // version 1.0
LPDIRECTDRAW7 lpdd7; // version 7.0

// create version 1.0 DirectDraw object interface
DirectDrawCreate(NULL, &lpdd, NULL);

// now look in DDRAW.H header, find IDIRECTDRAW7 interface
// ID and use it to query for the interface
lpdd->QueryInterface(IID_IDirectDraw7, &lpdd7);
```

At this point, you have two interface pointers, but you don't need the pointer to `IDIRECTDRAW`, so you should release it:

```
// release, decrement reference count
lpdd->Release();

// set to NULL to be safe
lpdd = NULL;
```

Remember that you should release any interface when you are done with it—so when your program terminates, you would also release the `IDIRECTDRAW7` interface:

```
// release, decrement reference count
lpdd7->Release();

// set to NULL to be safe
lpdd7 = NULL;
```

You might wonder whether there's a helper function to get a DirectDraw 7.0 interface. Well, actually there is, but I needed a good example to show off `QueryInterface()`, so don't be mad at me for not telling until now! Anyway, here's the easy way to get a DirectDraw 7.0 interface:

```
LPDIRECTDRAW7 lpdd7; // version 7.0 interface

// request the interface
DirectDrawCreateEx(NULL, (void **)&lpdd, IID_IDirectDraw7, NULL);
```

The function almost looks identical to the DirectDraw 1.0 version, but it has slightly different parameters, as shown in the following:

```
HRESULT WINAPI DirectDrawCreateEx(
 GUID FAR *lpGUID, // the GUID of the driver, NULL for active display
 LPVOID *lplpDD,  // receiver of the interface
```

Playrix Ex. 1023, page 138

04 0672318350 CH02 05•02•2003 2:33 PM Page 114


```
 REFIID iid,    // the interface ID of the interface you are requesting
 IUnknown FAR *pUnkOuter // advanced COM, NULL
);
```

That's all there is to using DirectX and COM. Of course, you haven't seen all the hundreds of functions that DirectX components have and all the interfaces, but you will one way or another <BG>.

Summary

At this point, you should have a handle on very basic Windows programming, and at least what DirectX is and its relationship to Windows. The cool thing is that you really don't need to know too much about Windows or DirectX (or all the versions of it) to get a lot out of this book, because in the next chapter we are going to build a virtual computer of sorts to run our 3D experiments and write code on. This way, you are insulated from the underlying mechanics of Win32/DirectX, and can focus on the 3D part. Win32/DirectX is nice to know, so if you want more info, try *Tricks of the Windows Game Programming Gurus*, along with any good Windows programming book.

Playrix Ex. 1023, page 139

CHAPTER **3**

3D Game Programming with a Virtual Computer

"NOPEACE..."

—Captured alien, *Independence Day*

In this chapter, we're basically going to build up a virtual computer system based on a software interface. It will support a linear 8- or 16-bit frame buffer (double buffered) and input devices, along with sound and music capabilities. With this interface, we can focus the remainder of the book on 3D math, graphics, and game programming. Here's what's in store:

- The design of a virtual computer graphics interface
- Building a Windows game console
- The API listing of the enhanced `T3DLIB` library from *Tricks of the Windows Game Programming Gurus*
- Implementing the virtual computer with `T3DLIB`
- The final game console
- Using the `T3DLIB` game library

IN THIS CHAPTER

- Introduction to the Virtual Computer Interface
- Building the Virtual Computer Interface
- The `T3DLIB` Game Console
- The `T3DLIB1` Library
- The `T3DLIB2` DirectX Input System
- The `T3DLIB3` Sound and Music Library
- The DirectMusic API Rapper
- Building the Final T3D Game Console
- Sample `T3LIB` Applications

Introduction to the Virtual Computer Interface

The goal of this book is to teach 3D graphics and game programming. However, my dilemma as an author is how to focus the book on just that, without talking about all the low-level details of Win32 programming, DirectX, and so on. Even though you should have some idea of Win32 programming and DirectX after reading the last chapter, it's still a

05 0672318350 CH03 05•02•2003 2:34 PM Page 116

bummer. Alas, my solution is to create a "virtual computer" based on the engine I developed in *Tricks of the Windows Game Programming Gurus* that you will simply use as a black box, so we can focus on the 3D graphics and game programming part. However, because of the sheer size of the engine, we still have a lot to discuss.

NOTE

This is basically the approach that OpenGL takes. You use GL libraries and/or add-ons that handle the low-level device-specific interfacing and housekeeping of whatever system you're on, including opening windows and getting input.

Ultimately, this really makes a lot of sense for a number of reasons. As long as you can communicate with a double-buffered, linearly-addressable graphics system that has support for input, sound, and music, who cares how it works? We are interested in the 3D graphics programming, not the low-level setup. Because 99% of our work is going to be related to rasterization, texture mapping, lighting, hidden surface removal, and so forth, this approach is feasible. So, all we really need is a black screen, the capability to talk to the joystick, keyboard, and mouse, and to maybe play some sound effects and music—right?

Therefore, instead of trying to explain every single detail of Win32, the DirectX foundation, and all that stuff (which I did in the first *Tricks*), we are simply going to use the API I wrote for that book as a tool to create a generic virtual computer on which we can run our 3D graphics experiments and write our games. Sound like a planetoid?

Using this layer of indirection, most of the 3D code will be generic enough that you could port it to Mac or Linux with very little work. All you would have to do is emulate the low-level interfaces, such as the double-buffered graphics system, sound, music, and input devices. The algorithms will stay the same for all the 3D code.

The only drawback to using the engine from the first *Tricks* is that a few of the data structures, and the design itself, were predicated on DirectX itself. I wanted the thinnest layer to DirectX for performance reasons. Hence, we could write another layer of software on top of my layer (which has some DirectX-centric parts to it) and totally be machine-independent, but I don't think it's worth the time. Bottom line is, if you want to port the 3D stuff, and as long as you can create a double-buffered display, everything will work. Sound, music, and input can be completely thrown away, because my functions make nothing more than a call to the DirectX functions anyway.

However, the thing to remember is that as long as you understand the API that I give you, you really do *not* need to know anything about Win32 or DirectX, because everything we do with 3D graphics simply consists of us, a frame buffer, and C/C++ code.

The fun part about this book is that we are only going to focus on 3D graphics and game programming. We are going to literally forget about all the details of setting up DirectX, getting input, and making sounds, and simply call API functions. We are going to

Playrix Ex. 1023, page 141

05 0672318350 CH03 05•02•2003 2:34 PM Page 117

accomplish this feat by creating a *virtual*, or *abstract*, graphics computer specification, and then implement it with the function API from the first *Tricks*. However, the main point of the virtual computer is to help us focus on 3D and not the details of the setup. With that in mind, here's the feature list you need to write a 3D game on any computer in the world:

1. Be able to create a window or screen on the display with a desired bit depth that is linearly addressable as a 2D matrix of pixels. Additionally, have support for an offscreen page or double buffer, so that images can be rendered offscreen and then copied or "flipped" to the primary display to create smooth animation.
2. Be able to get input from the system input devices such as the keyboard, mouse, and joystick. The input should be fairly raw and in a simple format.
3. **(Optional)** Be able to load and play sounds and music in common formats such as .WAV and .MID.

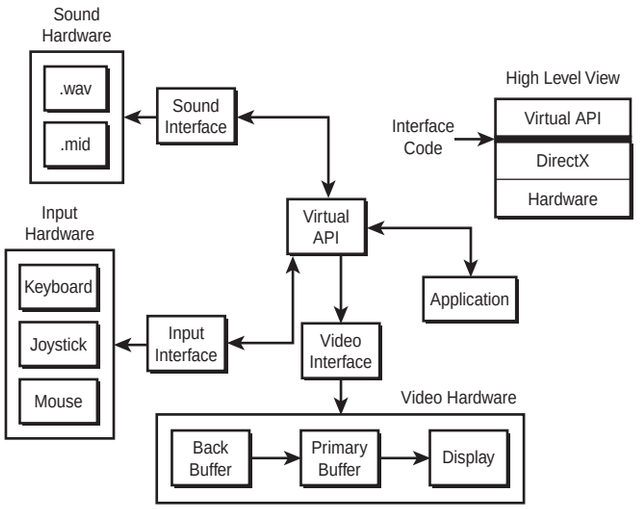


FIGURE 3.1 Systems-level diagram of the virtual computer.

Figure 3.1 illustrates a systems-level diagram of the virtual computer we want to design with software. I used to say, "If I can plot a pixel and read the keyboard, I can write *Doom*." It's true, all you need is the capability to access the frame buffer, and then everything else is simple. Moreover, because this book is about software 3D programming and rasterization, none of our algorithms will use any form of 3D acceleration. We are going to draw the polygons, compute the lighting, and everything else ourselves pixel-by-pixel in the frame buffer(s).

Playrix Ex. 1023, page 142

05 0672318350 CH03 05•02•2003 2:34 PM Page 118

NOTE

You might be saying, why use software when hardware is available? The answer is threefold: First, to be a great graphics programmer, you need to know how to do it yourself; second, knowing how things work helps you better understand acceleration; and last but not least, who will write the hardware code?

Building the Virtual Computer Interface

Building the virtual computer interface is rather easy if you have a functional library that can implement features (1) through (3) from the preceding section—which of course we do. That is, the library from the first *Tricks*. But before we get into really specific details, let's design a "model" of the functions and data structures that we would like to have in the virtual computer interface. Of course, this is just an example, and is going to be very high-level. Moreover, I'm just going to make the function names up as we go, so we have something to refer to. Alas, the final version of the virtual computer will be different, because when implementing it, we must take details into consideration. However, the following exercise will give you an idea of the big picture.

The Frame Buffer and Video System

Before we get started, let's assume that there is a way to initialize the video system and open a window with a specific resolution and bit depth. Let's call it `Create_Window()`, and here's what it might look like:

```
Create_Window(int width, int height, int bit_depth);
```

To create a 640×480 display with 8-bits per pixel, you would make the following call:

```
Create_Window(640, 480, 8);
```

And to create a 800×600 display with 16-bits per pixel, you would make the following call:

```
Create_Window(800, 600, 16);
```

Remember, this is all academic, and the real code to create a window and initialize the system might be a set of functions, but you get the idea. In reality, the "window" might actually be a Windows window, or it might be full screen. So we know that we aren't going to get away this easy. But let's move on now assuming that we have a window.

As I said, we are interested in designing a system that is based on a primary display buffer that is visible, and a secondary offscreen buffer that is not visible. Both buffers should be linearly addressable, where one *word*, be it a `BYTE`, `WORD`, or `QUAD` represents a single pixel, depending on the bit depth. Figure 3.2 represents our frame buffer system.

Playrix Ex. 1023, page 143

05 0672318350 CH03 05•02•2003 2:34 PM Page 119

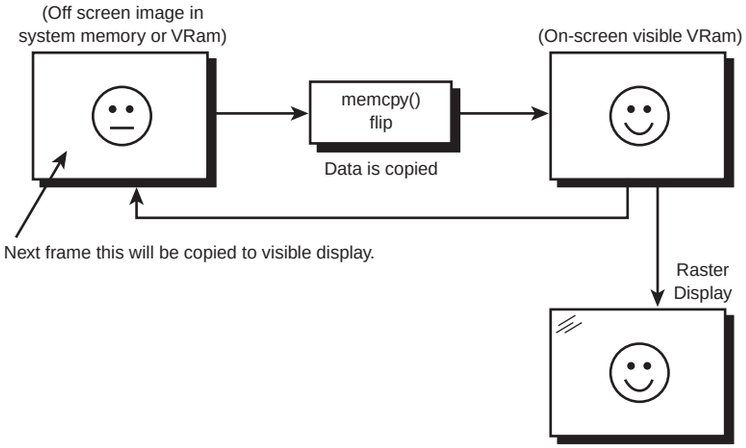


FIGURE 3.2 The frame buffer system.

You will notice that there is a *memory padding* region in the frame buffer memory in each of the primary and secondary display buffers. This models the fact that some video systems might be linear per row, but jump addresses row to row. For example, in Figure 3.3, we see this up close. In the example, the video system is set to 640×480×8, therefore, there is 1 byte per pixel, and there should be a total of 640 bytes per video line. But there's not! In this example, there are 1024 bytes per video line. This is a manifestation of the video card memory addressing, and is very common for many video cards. Hence, we need to model this discrepancy with what is commonly referred to as *memory pitch*.

Memory pitch models the fact that a video card might have extra memory per line because of caches or hardware addressing. This isn't a problem as long as we don't assume that memory pitch equals the *video pitch* (in the case of our example, 640 bytes = 640×1 byte per pixel). To locate a pixel on an 8-bit per pixel video system, we use the following code:

```
UCHAR *video_buffer; // ptr to the video buffer
int x,y;       // coordinates to the pixel
int memory_pitch;  // number of bytes per line

video_buffer[x + y*memory_pitch] = pixel;
```

On a 16-bit system, we would use slightly different code:

```
USHORT *video_buffer; // ptr to the video buffer
int x,y;       // coordinates to the pixel
int memory_pitch;   // number of bytes per line

video_buffer[x + y*(memory_pitch >> 1)] = pixel;
```

Playrix Ex. 1023, page 144

05 0672318350 CH03 05•02•2003 2:34 PM Page 120

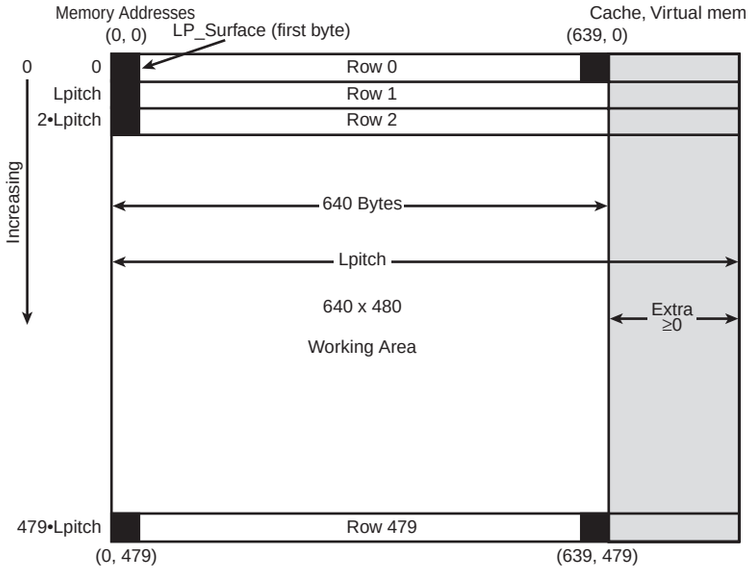


FIGURE 3.3 Close-up view of the hardware frame buffer.

Of course, if `memory_pitch` were the number of `USHORT`s per line, the bit shift operator >> used to divide by 2 wouldn't be needed.

With that in mind, let's create a model of the primary and secondary buffers. Remember, this is just a model, and the details will change when we actually implement the model with the `T3DLIB` game engine. Anyway, it looks like we need a pointer to both the primary and secondary buffers, along with variables to hold the memory pitch. For example, here's some globals that do the job:

```
UCHAR *primary_buffer; // the primary buffer
int  primary_pitch; // the memory pitch in bytes

UCHAR *secondary_buffer; // the secondary buffer
int  seconday_pitch;  // the memory pitch in bytes
```

Notice that both pointers are `UCHAR*`, and that the memory pitch is in terms of bytes? That's great if we are working with an 8-bit mode, but what about 16-bit modes, where there are 2 bytes per pixel, and each word is a `USHORT`? The answer is that I want to keep the data types simple and homogeneous. If you are working in a 16-bit mode, you can simply cast the pointers to `USHORT*`, or whatever. The point is that keeping the pointers `UCHAR*` is cleaner, along with always knowing memory pitch is in terms of bytes. This way, all the functions we write will have the same pointer/pitch parameter footprint.

Playrix Ex. 1023, page 145

05 0672318350 CH03 05•02•2003 2:34 PM Page 121

Locking Memory

Alright, we are getting there, but there is one other feature that we should work in—*locking the memory*. Many video cards have special memory that might be multi-ported, cached, or whatever. What this means is that when you access the memory, and you're reading and writing to and from the frame buffer(s), you must let the system know that you are accessing the memory, so that the system does not alter it during this time. When you're done, you unlock and release the memory and the system can go about its business.

TIP

If you're an old Windows programming guru from the 80s and early 90s, you should feel right at home, because in the earlier versions of Windows (1.0, 2.0, and 3.0), you had to lock memory—yuck!

All this means to us as programmers is that the pointers `primary_buffer` and `secondary_buffer` are only valid during a locked period. Furthermore, you can't assume that the addresses will be the same in the next lock. That is, maybe during one lock-unlock period the primary buffer was at

0x0FFF**E**FF**C**00000000

and during the next lock it's at

0x0FFF**F**FF**D**00000000

This is a manifestation of the hardware. It might move the frame buffer(s), so watch out! In any case, the locking procedure is as follows:

1. Lock the buffer of interest (either primary or secondary) and retrieve the starting memory address, along with the memory pitch.
2. Manipulate the video memory.
3. Unlock the buffer.

And of course, in 99% of the cases you will only lock-Read/Write-unlock the secondary buffer rather than the primary buffer, because you wouldn't want the user to "see" you altering the primary display buffer.

Based on this new feature, let's create a few functions that perform the locking and unlocking operations:

```
Lock_Primary(UCHAR **primary_buffer, int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);

Lock_Secondary(UCHAR **secondary_buffer, int *secondary_pitch);
Unlock_Secondary(UCHAR *secondary buffer);
```

Playrix Ex. 1023, page 146

05 0672318350 CH03 05•02•2003 2:34 PM Page 122

Basically, to lock a buffer you call the function with the addresses of the storage for the frame buffer and memory pitch. The function then locks the surface and alters the sent variables (writes to them). Then you are free to use them. When you're done, you unlock the surface with a call to `Unlock_*()` with a pointer to the locked surface. Simple, huh?

As an example, let's see how we would write a single pixel in 800×600 mode to the screen of the secondary buffer in both 8-bit mode and 16-bit mode. First, here's the 8-bit example:

```
UCHAR *primary_buffer; // the primary buffer
int  primary_pitch; // the memory pitch in bytes

UCHAR *secondary_buffer; // the secondary buffer
int  seconday_pitch;  // the memory pitch in bytes

UCHAR pixel; // the pixel to write
int x,y;   // coordinates of pixel

// step 0:create the window
Create_Window(800, 600, 8);

// step 1: lock the secondary surface
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// write the pixel to the center of the screen
secondary_buffer[x + y*secondary_pitch] = pixel;

// we are done, so unlock the secondary buffer
Unlock_Secondary(secondary_buffer);
```

That was easy! Now here's the same code, assuming that the system is in 16-bit mode or 2 bytes per pixel.

```
UCHAR *primary_buffer; // the primary buffer
int  primary_pitch; // the memory pitch in bytes

UCHAR *secondary_buffer; // the secondary buffer
int  seconday_pitch;  // the memory pitch in bytes

USHORT pixel; // the pixel to write, 16-bits
int x,y;   // coordinates of pixel

// step 0:create the window
Create_Window(800, 600, 16);
```

Playrix Ex. 1023, page 147

05 0672318350 CH03 05•02•2003 2:34 PM Page 123


```
// step 1: lock the secondary surface
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// at this point, we have to be careful since the
// locked pointer is a UCHAR * and not a USHORT *
// hence we need to cast it

USHORT *video_buffer = (USHORT *)secondary_buffer;

// write the pixel to the center of the screen
video_buffer[x + y*(secondary_pitch >> 1)] = pixel;

// we are done, so unlock the secondary buffer
Unlock_Secondary(secondary_buffer);
```


Note that we also had to divide the memory pitch by 2 (using the >> operator) because it's in terms of bytes, and if we want to use the built-in pointer arithmetic properly, we need to convert the memory pitch to the number of `USHORT`s per line. Of course, there are a hundred ways you can write this code, but you get the point. Get it—point? You could say I have a one-dimensional sense of humor <GRIN>.

Finally, there is one detail that we have left out—the 8- or 16-bit pixel data format. Oops!

Working with Color

We are primarily going to work with 8-bit palettized video modes and 16-bit RGB modes in our 3D game programming for speed reasons. In 8-bit mode, there is 1 byte per pixel, and in 16-bit mode, there are 2 bytes per pixel. However, the encoding is completely different.

8-Bit Color Mode

8-bit mode uses a standard 256-entry *color lookup table (CLUT)*, as shown in Figure 3.4. 16-bit mode uses a standard RGB encoding. When working with the 8-bit mode, we must fill the color lookup table with RGB values for each color entry in the table for 0…255. We assume that there is a function(s) to access this table and read/alter each entry, so let's not worry about it for now.

I just want you to know that there aren't any surprises about 8-bit mode—it's the standard 1 byte per pixel system, where each pixel value is an index into the color lookup table, and each table entry is composed of a color triad with 8-bits for each channel, (red, green, blue), for a total of 24-bits per entry.

TIP

Some cards only use 6 bits of each 8-bit channel entry, or in other words, there are only $2^6 = 64$ shades of each primary—red, green, blue—rather than the standard $2^8 = 256$ shades, with a full 8 bits per channel color lookup.

Playrix Ex. 1023, page 148

05 0672318350 CH03 05•02•2003 2:34 PM Page 124

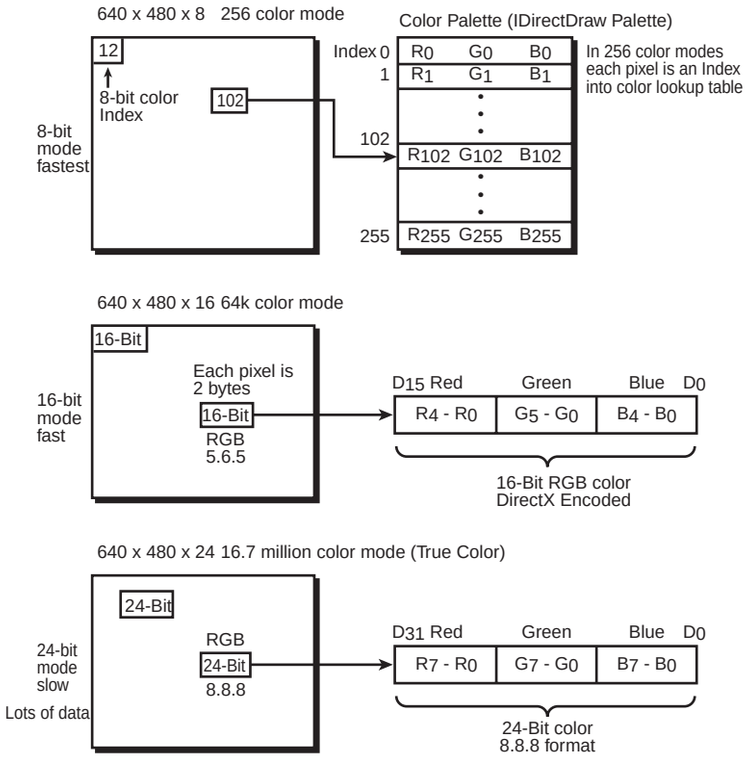


FIGURE 3.4 Comparisons of various color depths and their implementations.

16-Bit Color Mode

Working with 16-bit color is a little simpler than 8-bit—sorta. In the case of 16-bit color, there isn't a lookup table anymore. In fact, each pixel is in the form of a concatenated bit string word of RGB. In this form, the pixel format is either 5.5.5 (RGB) or 5.6.5 (RGB) format; that is, 5 bits of red, 5 bits of green, 5 bits of blue, or 5 bits of red, 6 bits of green, and 5 bits of blue, respectively. This is shown in Figure 3.5. This RGB pixel format concept sometimes gives people a lot of trouble (I have thousands of emails asking about it). The common question is how to build a pixel up? I think that the problem is that everyone thinks it's more complex than it really is. There is nothing to it really, just some bit manipulations.

In 5.5.5 RGB mode, there are 5 bits for every pixel, or $2^5 = 32$ intensities for each color channel. Therefore, a single pixel's RGB values each range from 0–31 (5 bits each), and there are a total of 32*32*32 = 32,768 different colors possible. Hence, all you need to do is write some code that takes a triad of three values. Suppose you have `r,g,` and `b`, each

Playrix Ex. 1023, page 149

05 0672318350 CH03 05•02•2003 2:34 PM Page 125

ranging from 0–31 (31 being the highest intensity) and you use bit manipulation operators to build the pixel up. Here's one such macro for 5.5.5 mode:

```
// this builds a 16 bit color value in 5.5.5 format (1-bit alpha mode)
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))
```

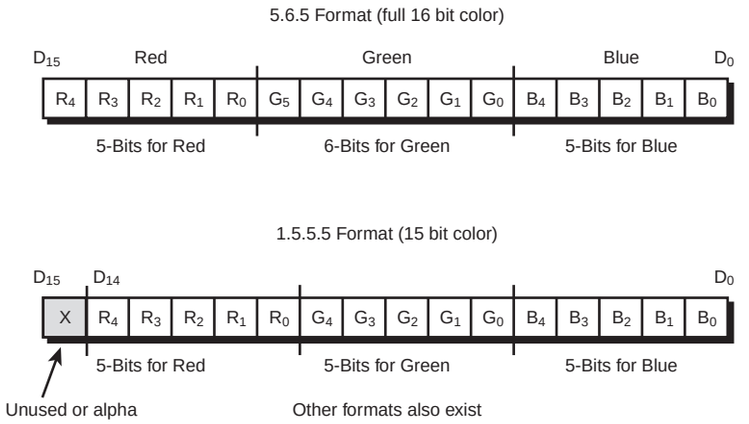


FIGURE 3.5 16-bit color encodings for both the 5.5.5 and 5.6.5 formats.

Similarly, in 5.6.5 mode, there is an extra bit for green, so the range of RGB pixel values is 0–31 for red, 0–63 for green, and 0–31 for blue. This gives us a total of 32*64*32 = 65,536 possible colors. Once again, here's a macro to build a pixel from a triad of values `r`,`g`, and `b`:

```
// this builds a 16 bit color value in 5.6.5 format (green dominate mode)
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))
```

This problem has given people so much trouble in the past that we are going to work some logic into the system to take this into account. The graphics system, when selected into 16-bit mode, will determine the pixel format (either 5.5.5 or 5.6.5) and set flags and function pointers, so you can always use the same macro to build up a 16-bit color pixel. However, this feature will be in the "real" library we make later. For now, just make note of this detail, because I don't want to hear about how everything "looks green" when someone assumes a pixel format. Now, let's talk about animating the display.

Animating the Display

The final bit of functionality we need to implement is a way to flip the display, or copy the secondary buffer to the primary buffer, as shown in Figure 3.6. Of course, we could simply lock both surfaces and do a line-by-line copy of memory. However, many cards have special hardware to flip the display and copy them faster than we can with the processor. Hence, we will allow for this option and use a black box approach to flipping

Playrix Ex. 1023, page 150

05 0672318350 CH03 05•02•2003 2:34 PM Page 126

the display via an API call. Let's call the function `Flip_Display()`. It takes no parameters, and simply copies the secondary buffer to the primary buffer (with software or hardware, but we don't know which).

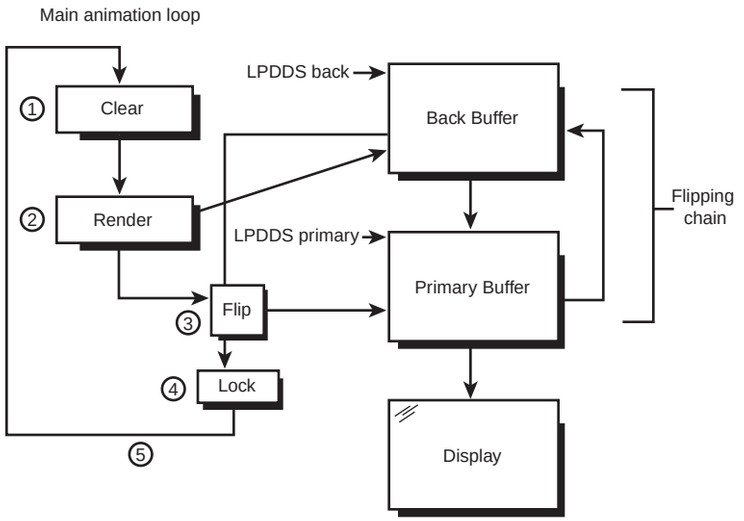


FIGURE 3.6 A page-flipped animation system.

CAUTION

You might wonder why I said to copy the secondary buffer line by line, rather than use a `memcpy()` function. This is because the secondary buffer may or may not be contiguous from line to line, and `memcpy()` has no way of knowing this. Alas, you must copy the buffers line by line, and then address each row by the appropriate memory pitch if you want to do it yourself.

However, there is a rule we are going to follow—when you make a call to `Flip_Display()`, you must make sure that both the primary and secondary buffers are *unlocked*. If they aren't, the flip won't occur, and an error will be thrown—or you will get some exercise with your index finger pressing the reset button on your computer. Therefore, before your call to `Flip_Display()`, you should always make sure that both buffers are unlocked and that the system has access to them.

Additionally, there are two more helper functions that come in handy when performing animation: functions to *clear* the buffers. You could once again just lock the buffers and use `memset()` or something to clear the buffer(s) line by line, but many cards have hardware-fill capabilities to clear memory buffers. Let's assume that where hardware filling is available it will be used; otherwise, software will be used to fill the memory. In either case,

Playrix Ex. 1023, page 151

05 0672318350 CH03 05•02•2003 2:34 PM Page 127

we don't care, because we will make up API functions to do this for both the secondary and primary buffers:

```
Fill_Primary(int color);
Fill_Secondary(int color);
```

The functions assume that the buffers are unlocked and the `color` parameter would be either 8- or 16-bit, depending on the mode. In most cases, you want to clear the buffer with black, so you would use `0`. This is because most palettes use index `0` as black, and the `USHORT` value of `0` is equivalent to RGB (0,0,0), which is black, also.

Using the clear function is simple. At the top of your animation loop, you would make a call to clear the screen out. However, because you are going to later copy the secondary buffer to the primary buffer at the end of the animation cycle, there is no need to clear the primary buffer, so all you need to do is clear the secondary buffer.

As an example, here's how you would draw an animated display of randomly positioned pixels in 800×600×8 mode, at 30 fps:

```
UCHAR *primary_buffer; // the primary buffer
int  primary_pitch; // the memory pitch in bytes

UCHAR *secondary_buffer; // the secondary buffer
int  seconday_pitch;  // the memory pitch in bytes

UCHAR pixel; // the pixel to write, 8-bits
int x,y;   // coordinates of pixel

// step 0:create the window
Create_Window(800, 600, 8);

// enter infinite loop
while(1)
{
// clear the secondary buffer
Fill_Secondary(0);

// step 1: lock the secondary surface
Lock_Secondary(&secondary_buffer, &secondary_pitch);

for (int num_dots=0; num_dots < 1000; num_dots++)
  {
  // get a random x,y,pixel
  x = rand()%800;
```

Playrix Ex. 1023, page 152

05 0672318350 CH03 05•02•2003 2:34 PM Page 128


```
  y = rand()%600;
  pixel = rand()%256;

  // write the pixel to the center of the screen
  video_buffer[x + y*secondary_pitch)] = pixel;
  } // end for num_dots

// we are done, so unlock the secondary buffer
Unlock_Secondary(secondary_buffer);

// flip the display
Flip_Display();

// wait a sec
Sleep(33);

} // end while
```

Of course, the loop is infinite, and in a real Windows application, we can't be this greedy and not release each frame back to the main event loop—but you get the idea.

The Complete Virtual Graphics System

At this point, we have enough functionality to implement the virtual graphics system, so let's briefly summarize what we have thus far to make sure we all have the same quantum numbers:

- We assume that we can call functions that will set the graphics system up and open a window (it could be full-screen) with any size and bit depth that we desire. However, most of the time, we are only going to work with 8- and 16-bit modes for speed.
- The system is composed of both a primary and secondary offscreen display buffer. The buffers are linearly addressable, but have a memory pitch associated with them. Moreover, to access the actual memory of either buffer, the buffer must be locked. When the operation is complete, the buffer must be unlocked.
- To animate the display and "flip" one page to another, we assume there is a function that takes advantage of hardware blitting (blocking image transfers) acceleration to accomplish this. However, before making the call, we must make sure that both the primary and secondary buffers are unlocked. Also, there are buffer-clearing functions available that use hardware acceleration (if it's available).

Hopefully, you can see that if we can implement this virtual software system, the 3D graphics aspect of our programming is the only problem left. All we have to interface with to get our image on the screen is the frame buffers and a few API calls here and there.

Playrix Ex. 1023, page 153

05 0672318350 CH03 05•02•2003 2:34 PM Page 129

I/O, Sound, and Music

Because the idea of this book is to make 3D games that are playable, we need to be able to obtain input from the player, as well as play sounds and music. Otherwise, people are going to think you are weird when you use mind control to play the game along with your own sound effects <GRIN>.

Of course, all of this functionality is part of DirectX, and I have written wrappers around all of it to make it as painless as possible. But the point I want to make here is that even if you port the code from the book to another system that doesn't support DirectX, you have very little work. The engine from the first *Tricks* really only has the following fundamental capabilities (in order):

1. Initialize and detect all input devices: keyboard, mouse, and joystick.
2. Read the data or *state* of all input devices, and put the data in simple data structures.
3. Initialize the sound and music system.
4. Load a sound off disk in .WAV format and play it once or loop it. Load a MIDI song off disk and play it once or loop it. Also be capable of testing the status of sounds and music when playing.
5. Shut everything down.

The preceding list is the exact functionality that we will implement. However, if you need to, you can simply "rip" all the I/O and sound code from any of the demos and port them with just the graphics interface. Of course, you need some kind of input, even if it is `getch()`!

As a sample API that looks something like what we will end up with, you might create something like this:

```
// this initializes all the input devices
Init_Input_Devs();

// this shuts down the input system
Shutdown_Input_Devs();

// this reads the keyboard, where data is some structure that
// holds the keyboard state, maybe an array
Read_Keyboard(&data);

// this reads the mouse, where data is some structure that
// holds the mouse state, maybe the position and buttons
Read_Mouse(&data);
```

Playrix Ex. 1023, page 154

05 0672318350 CH03 05•02•2003 2:34 PM Page 130


```
// this reads the joystick, where data is some structure that
// holds the joystick state, maybe the position and buttons
Read_Joystick(&data);

// this initializes the sound and music system
Init_Sound_Music();

// this shuts down the sound and music system
Shutdown_Sound_Music();

// this loads a .WAV file and returns an id
int id = Load_WAV(char *filename);

// this plays a .WAV file based on an id
Play_WAV(int id);

// this loads a .MID file and returns an id
int id = Load_MID(char *filename);

// plays a .MID file based on an id
Play_MID(int id);
```

If we (you) can implement these functions, that's all you need for the I/O and sound interfaces.

Well, that's about it for the abstract design of the virtual computer interface. Remember, this was just an exercise in software design. Now we need to bang out the details—that is, the actual implementation of all the functions—I can't wait!

TIP

What we just did is actually a very useful exercise in design. We basically designed a portable graphics system. This is exactly how you would go about designing a game engine that you can port to other platforms just by implementing the "insides" of a very small set of functions.

The T3DLIB Game Console

At this point, I have another dilemma brewing. I want to show you the generic Windows Game Console prototype we are going to use to shell our games. However, within it are calls to the API. Should I show you the library function API now, or should I wait? Well, I'm in a top-down mood right now, and because you are an intermediate or advanced game programmer, I think you can handle waiting. If there is anything that you can't follow, simply take a peek at the Library API listings and descriptions.

Playrix Ex. 1023, page 155

05 0672318350 CH03 05•02•2003 2:34 PM Page 131

T3DLIB System Overview

Based on our discussion of the virtual computer design, our goal is to abstract the Win32/DirectX model into a very simple double-buffered graphics system that we can access along with support for input and sound. Along the lines of this goal, I have created what's called the *T3D Game Console*. The Game Console is really the first step in creating the virtual computer interface. Before we can get into DirectX, input, and sound, first we have to remove Windows from the equation. How I'm going to do this is by creating a template, or Game Console as I call it, around Windows, so that from our point of view, a Windows application looks like a standard DOS/Unix application with a `main()` and that's it.

What we're going to do is build up the Game Console layer by layer until we have our virtual computer. Let's first start with removing Windows from the equation as best we can.

The Basic Game Console

The first iteration of the Game Console should perform the following tasks:

1. Open a window.
2. Call a user-defined initialization function: `Game_Init()`.
3. Enter into the main Windows event loop, process any messages for you, and return.
4. Call a user-defined "main" work function `Game_Main()` that performs one cycle of game logic and then returns.
5. Loop to step 3 until the user kills the application.
6. Call a user-defined shutdown function `Game_Shutdown()` that cleans up.

Figure 3.7 depicts the flow diagram of the Game Console. You will notice that I have decided to call the initialization, runtime, and shutdown functions `Game_Init()`, `Game_Main()`, and `Game_Shutdown()`, respectively.

If we can implement a shell program that performs the functionality of steps 1 through 6, we can completely forget about Windows and focus on the functions `Game_Init()`, `Game_Main()`, and `Game_Shutdown()` only. First, let's prototype out the three functions with some generic interfaces:

```
// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);
```

Playrix Ex. 1023, page 156

05 0672318350 CH03 05•02•2003 2:34 PM Page 132

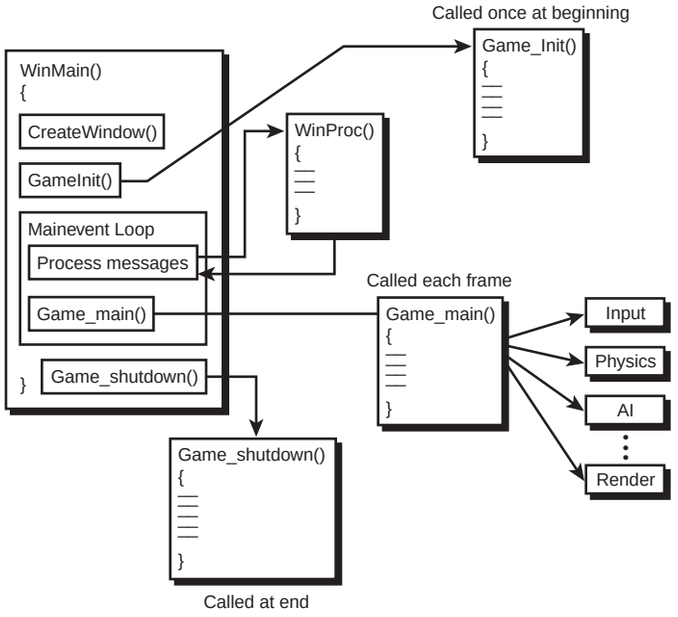


FIGURE 3.7 The control flow of the Game Console.

Not much to them, huh? Basically, I have written them so you can (if you want) send a structure to the functions, but I find I never use the feature. In any case, with that in mind, we need to write a Windows program shell program that does the following:

1. Creates a Windows class and registers the class.
2. Creates a window.
3. Has an event loop and a `WinProc()` that handles the basic messages `WM_CREATE`, `WM_PAINT`, and `WM_DESTROY`.
4. Before the event loop is called, a call to `Game_Init()` is called, so you can do your initialization here.
5. During the event loop, `Game_Main()` is called each cycle, so your game can process and run.
6. When the Window is terminated, a call to `Game_Shutdown()` is made, so you can clean up.

The following is a `T3DCONSOLEALPHA1.CPP` from the CD, a complete Windows program that does all of these steps. I have highlighted the function calls to our worker functions:

Playrix Ex. 1023, page 157

05 0672318350 CH03 05•02•2003 2:34 PM Page 133


```
// T3DCONSOLEALPHA1.CPP - First alpha iteration of game console

// INCLUDES ///////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN

#include <windows.h>  // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>


// DEFINES ////////////////////////////////////////////////

// defines for windows interface
#define WINDOW_CLASS_NAME "WIN3DCLASS" // class name
#define WINDOW_TITLE   "T3D Graphics Console Alpha 1.0"
#define WINDOW_WIDTH   320 // size of window
#define WINDOW_HEIGHT   240

// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)  ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// PROTOTYPES /////////////////////////////////////////////

// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// GLOBALS ////////////////////////////////////////////////
```

Playrix Ex. 1023, page 158

05 0672318350 CH03 05•02•2003 2:34 PM Page 134


```
HWND main_window_handle  = NULL; // save the window handle
HINSTANCE main_instance  = NULL; // save the instance
char buffer[256];         // used to print text


// FUNCTIONS ////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
         UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT  ps;    // used in WM_PAINT
HDC    hdc;    // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
  // do initialization stuff here
  return(0);
  } break;

    case WM_PAINT:
    {
    // start painting
    hdc = BeginPaint(hwnd,&ps);

    // end painting
    EndPaint(hwnd,&ps);
    return(0);
    } break;

  case WM_DESTROY:
    {
  // kill the application
  PostQuitMessage(0);
  return(0);
  } break;

  default:break;
```

Playrix Ex. 1023, page 159

05 0672318350 CH03 05•02•2003 2:34 PM Page 135


```
  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////

int WINAPI WinMain(  HINSTANCE hinstance,
      HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow)
{
// this is the winmain function

WNDCLASS winclass;  // this will hold the class we create
HWND   hwnd;    // generic window handle
MSG   msg;   // generic message
HDC   hdc;  // generic dc
PAINTSTRUCT ps; // generic paintstruct

// first fill in the window class stucture
winclass.style    = CS_DBLCLKS ¦ CS_OWNDC ¦
             CS_HREDRAW ¦ CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra   = 0;
winclass.cbWndExtra   = 0;
winclass.hInstance   = hinstance;
winclass.hIcon     = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor    = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;

// register the window class
if (!RegisterClass(&winclass))
  return(0);

// create the window, note the test to see if WINDOWED_APP is
// true to select the appropriate window flags
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
       WINDOW_TITLE,   // title
```

Playrix Ex. 1023, page 160

05 0672318350 CH03 05•02•2003 2:34 PM Page 136


```
            WS_OVERLAPPED ¦ WS_SYSMENU ¦ WS_CAPTION,
       0,0,    // x,y
       WINDOW_WIDTH, // width
            WINDOW_HEIGHT, // height
       NULL,    // handle to parent
       NULL,    // handle to menu
       hinstance,// instance
       NULL)))  // creation parms
return(0);

// save the window handle and instance in a global
main_window_handle = hwnd;
main_instance   = hinstance;

// make sure window is visible
ShowWindow(main_window_handle, SW_SHOW);

// perform all game console specific initialization
Game_Init();

// enter main event loop
while(1)
  {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // test if this is a quit
     if (msg.message == WM_QUIT)
       break;

    // translate any accelerator keys
    TranslateMessage(&msg);

    // send the message to the window proc
    DispatchMessage(&msg);
     } // end if

  // main game processing goes here
  Game_Main();
  } // end while

// shutdown game and release all resources
Game_Shutdown();
```

Playrix Ex. 1023, page 161

05 0672318350 CH03 05•02•2003 2:34 PM Page 137


```
// return to Windows like this
return(msg.wParam);

} // end WinMain

// T3D II GAME PROGRAMMING CONSOLE FUNCTIONS ////////////////

int Game_Init(void *parms)
{
// this function is where you do all the initialization
// for your game

// return success
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// this function is where you shutdown your game and
// release all resources that you allocated

// return success
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for your game go here!

// game logic here...

// check if user is trying to exit
if (KEY_DOWN(VK_ESCAPE))
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);
  } // end if
```

Playrix Ex. 1023, page 162

05 0672318350 CH03 05•02•2003 2:34 PM Page 138


```
// return success
return(1);

} // end Game_Main
```

The name of the source code on the CD is `T3DCONSOLEALPHA1.CPP`, and the executable is, of course, `T3DCONSOLEALPHA1.EXE`. If you run the .EXE, all you will see is the image in Figure 3.8—basically a little window with nothing happening. However, behind the scenes a lot is really going on: a window was created, `Game_Init()` was called, and `Game_Main()` is being called over and over each event cycle. Finally, when you close the window, `Game_Shutdown()` is called.

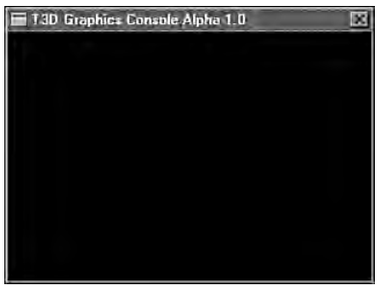


FIGURE 3.8 Alpha game console `T3DCONSOLEALPHA1.EXE` in action.

So all you need to do is put your entire game functionality within those three functions. That's all there is to it—we have abstracted a Windows application to three function calls, and everything else is done for us.

NOTE

It's really too bad that most of the books on Windows programming make it so hard. It really isn't—and it's kind of fun!

Based on this first version of the Game Console, we have all the Windows functionality that we need. Now we need the DirectX interface that emulates our virtual computer for us and gives us the basic double-buffered graphics system with sound and input capabilities. At this point, we could dive right in and take a look at the final Game Console that does all this, but I think it's a better idea to take a look at the API that I am going to use to build it up. Now remember, you don't need to completely understand the API and all the functions. Simply take a look at all of them and their associated explanations and examples.

Playrix Ex. 1023, page 163

05 0672318350 CH03 05•02•2003 2:34 PM Page 139

After you have reviewed the tools that we will use to build the final Game Console, we will take the T3DGAMECONSOLEALPHA1.CPP file and add to it in various areas, so that the results are the final virtual computer interface that we need. Then we will use this "template" as a starting point for all the demos and games in the book. Additionally, before ending the chapter, I will give a number of examples that show how to use the T3DLIB's graphics, sound, and input abilities. Alrighty then, let's move on to the three subsystems that make up the T3DLIB game engine from the first *Tricks*.

The T3DLIB1 Library

At this point, we're ready to take a look at all the #defines, macros, data structures, and functions that compose the graphics module API of the T3DLIB game library—T3DLIB1.CPP.

NOTE

If you have already read the first *Tricks*, you will notice that the engine is almost identical except for the additional 16-bit support sprinkled throughout, along with windowed support. However, the code is still compatible, so you can take the demos from the first *Tricks* and compile them with this new version of T3DLIB1.CPP and they will still work! And of course, everything will compile with DirectX 8.0 or 9.0+.

The module consists of two files: T3DLIB1.CPP¦H. Hence, you simply link these into your programs and then use the API.

The DirectX Graphics Engine Architecture

T3DLIB1 is a fairly simple 2D engine, as shown in Figure 3.9. Basically, it's a 2D, 8–16-bit color, doubled-buffered DirectX engine that has support for any resolution along with clipping. The engine also supports both windowed and full-screen displays, and takes care of all the setup. So, whether you're in a windowed display or full-screen mode, you always write to the offscreen or secondary buffer, and logic takes care of the detailed process of copying or flipping the secondary buffer to the primary display. This is done regardless of whether the primary display window is a window or full-screen.

To build an application using the library, you'll need to include T3DLIB1.CPP¦H, along with DDRAW.LIB (DirectDraw Library) and WINMM.LIB (Win32 Multimedia Library).

NOTE

You only need WINMM.LIB if you are using Visual C++.

Playrix Ex. 1023, page 164

05 0672318350 CH03 05•02•2003 2:34 PM Page 140

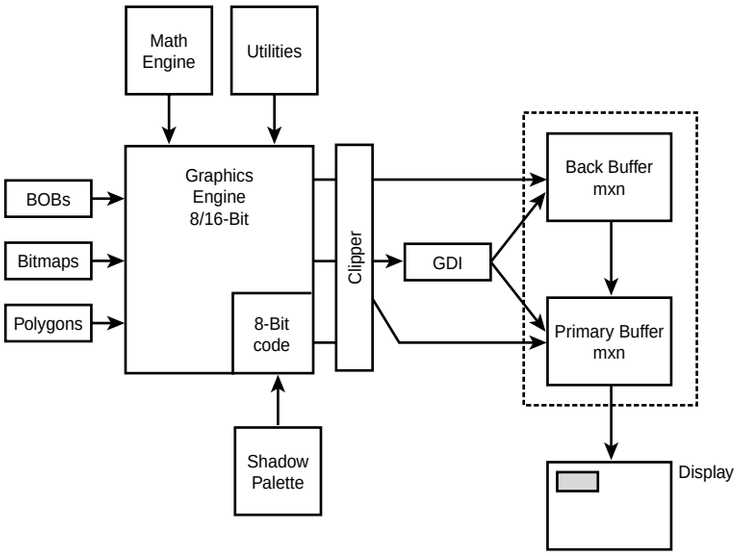


FIGURE 3.9 The architecture of the graphics engine.

Basic Definitions

The engine has one header file, `T3DLIB1.H`, and within are a number of `#defines` that the engine uses. Here they are for your reference:

```
// watch for multiple inclusions
#ifndef T3DLIB1
#define T3DLIB1

// DEFINES ////////////////////////////////////////////////

// default screen values, these are all overriden by the
// call to DDraw_Init() and are just here to have something
// to set the globals to instead of constant values
#define SCREEN_WIDTH    640 // size of screen
#define SCREEN_HEIGHT    480
#define SCREEN_BPP     8  // bits per pixel
#define MAX_COLORS_PALETTE 256


#define DEFAULT_PALETTE_FILE "PALDATA2.PAL"
```

Playrix Ex. 1023, page 165

05 0672318350 CH03 05•02•2003 2:34 PM Page 141


```
// used for selecting full screen/windowed mode
#define SCREEN_FULLSCREEN  0
#define SCREEN_WINDOWED   1

// bitmap defines
#define BITMAP_ID      0x4D42 // universal id for a bitmap
#define BITMAP_STATE_DEAD  0
#define BITMAP_STATE_ALIVE  1
#define BITMAP_STATE_DYING  2
#define BITMAP_ATTR_LOADED  128

#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS  1

// directdraw pixel format defines, used to help
// bitmap loader put data in proper format
#define DD_PIXEL_FORMAT8    8
#define DD_PIXEL_FORMAT555   15
#define DD_PIXEL_FORMAT565   16
#define DD_PIXEL_FORMAT888   24
#define DD_PIXEL_FORMATALPHA888 32

// defines for BOBs
#define BOB_STATE_DEAD     0  // this is a dead bob
#define BOB_STATE_ALIVE    1  // this is a live bob
#define BOB_STATE_DYING    2  // this bob is dying
#define BOB_STATE_ANIM_DONE  1  // done animation state
#define MAX_BOB_FRAMES     64  // maximum number of bob frames
#define MAX_BOB_ANIMATIONS   16  // maximum number of animation sequences

#define BOB_ATTR_SINGLE_FRAME  1  // bob has single frame
#define BOB_ATTR_MULTI_FRAME  2  // bob has multiple frames
#define BOB_ATTR_MULTI_ANIM   4  // bob has multiple animations
#define BOB_ATTR_ANIM_ONE_SHOT 8  // bob will perform the animation once
#define BOB_ATTR_VISIBLE    16 // bob is visible
#define BOB_ATTR_BOUNCE     32 // bob bounces off edges
#define BOB_ATTR_WRAPAROUND   64 // bob wraps around edges
#define BOB_ATTR_LOADED     128 // the bob has been loaded
#define BOB_ATTR_CLONE     256 // the bob is a clone

// screen transition commands (256 color modes only)
#define SCREEN_DARKNESS 0     // fade to black
#define SCREEN_WHITENESS 1     // fade to white
```

Playrix Ex. 1023, page 166

05 0672318350 CH03 05•02•2003 2:34 PM Page 142


```
#define SCREEN_SWIPE_X  2     // do a horizontal swipe
#define SCREEN_SWIPE_Y  3     // do a vertical swipe
#define SCREEN_DISOLVE  4     // a pixel disolve
#define SCREEN_SCRUNCH  5     // a square compression
#define SCREEN_BLUENESS 6     // fade to blue
#define SCREEN_REDNESS  7     // fade to red
#define SCREEN_GREENNESS 8     // fade to green

// defines for Blink_Colors
#define BLINKER_ADD      0  // add a light to database
#define BLINKER_DELETE    1  // delete a light from database
#define BLINKER_UPDATE    2  // update a light
#define BLINKER_RUN      3  // run normal

// pi defines
#define PI     ((float)3.141592654f)
#define PI2    ((float)6.283185307f)
#define PI_DIV_2  ((float)1.570796327f)
#define PI_DIV_4  ((float)0.785398163f)
#define PI_INV   ((float)0.318309886f)

// fixed point mathematics constants
#define FIXP16_SHIFT   16
#define FIXP16_MAG    65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

Working Macros

Next are all the macros we've written thus far. Again, you've seen them all in one place or another, but here they all are at once:

```
// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)  ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// this builds a 16 bit color value in 5.5.5 format (1-bit alpha mode)
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))

// this builds a 16 bit color value in 5.6.5 format (green dominate mode)
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))
```

Playrix Ex. 1023, page 167

05 0672318350 CH03 05•02•2003 2:34 PM Page 143


```
// this builds a 24 bit color value in 8.8.8 format
#define _RGB24BIT(a,r,g,b) ((b) + ((g) << 8) + ((r) << 16) )

// this builds a 32 bit color value in A.8.8.8 format (8-bit alpha mode)
#define _RGB32BIT(a,r,g,b) ((b) + ((g) << 8) + ((r) << 16) + ((a) << 24))

// bit manipulation macros
#define SET_BIT(word,bit_flag)  ((word)=((word) ¦ (bit_flag)))
#define RESET_BIT(word,bit_flag) ((word)=((word) & (~bit_flag)))

// initializes a direct draw struct,
// basically zeros it and sets the dwSize field
#define DDRAW_INIT_STRUCT(ddstruct) {
memset(&ddstruct,0,sizeof(ddstruct));ddstruct.dwSize=sizeof(ddstruct); }

// used to compute the min and max of two expresions
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define MAX(a, b) (((a) > (b)) ? (b) : (a))

// used for swapping algorithm
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// some math macros
#define DEG_TO_RAD(ang) ((ang)*PI/180)
#define RAD_TO_DEG(rads) ((rads)*180/PI)

#define RAND_RANGE(x,y) ( (x) + (rand()%((y)-(x)+1)))
```

Data Types and Structures

The next set of code elements are the types and data structures that the engine uses. Many of them aren't really of concern, because a lot of them are for the 2D aspect from the first *Tricks*, but it doesn't hurt to see all of it.

```
// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char UCHAR;
typedef unsigned char BYTE;
typedef unsigned int  QUAD;
typedef unsigned int  UINT;
```

Playrix Ex. 1023, page 168

05 0672318350 CH03 05•02•2003 2:34 PM Page 144


```
// container structure for bitmaps .BMP file
typedef struct BITMAP_FILE_TAG
{
BITMAPFILEHEADER bitmapfileheader; // contains the bitmapfile
➥header
BITMAPINFOHEADER bitmapinfoheader; //info including
➥the palette
PALETTEENTRY   palette[256];   // palette
➥stored here
UCHAR      *buffer;      // pointer to the data

    } BITMAP_FILE, *BITMAP_FILE_PTR;

// the blitter object structure BOB
typedef struct BOB_TYP
    {
    int state;     // the state of the object (general)
    int anim_state;   // an animation state variable, up to you
    int attr;      // attributes pertaining to the object (general)
    float x,y;     // position bitmap will be displayed at
    float xv,yv;    // velocity of object
    int width, height; // the width and height of the bob
    int width_fill;   // internal, used to force 8*x wide surfaces
    int bpp;      // bits per pixel
    int counter_1;   // general counters
    int counter_2;
    int max_count_1;  // general threshold values;
    int max_count_2;
    int varsI[16];   // stack of 16 integers
    float varsF[16];  // stack of 16 floats
    int curr_frame;   // current animation frame
    int num_frames;   // total number of animation frames
    int curr_animation; // index of current animation
    int anim_counter;  // used to time animation transitions
    int anim_index;   // animation element index
    int anim_count_max; // number of cycles before animation
    int *animations[MAX_BOB_ANIMATIONS]; // animation sequences

    // the bitmap images DD surfaces
    LPDIRECTDRAWSURFACE7 images[MAX_BOB_FRAMES];
    } BOB, *BOB_PTR;

// the simple bitmap image
typedef struct BITMAP_IMAGE_TYP
```

Playrix Ex. 1023, page 169

05 0672318350 CH03 05•02•2003 2:34 PM Page 145


```
    {
    int state;     // state of bitmap
    int attr;      // attributes of bitmap
    int x,y;      // position of bitmap
    int width, height; // size of bitmap
    int num_bytes;   // total bytes of bitmap
    int bpp;      // bits per pixel
    UCHAR *buffer;   // pixels of bitmap

    } BITMAP_IMAGE, *BITMAP_IMAGE_PTR;

// blinking light structure
typedef struct BLINKER_TYP
    {
    // user sets these
    int color_index;     // index of color to blink
    PALETTEENTRY on_color;  // RGB value of "on" color
    PALETTEENTRY off_color; // RGB value of "off" color
    int on_time;       // number of frames to keep "on"
    int off_time;      // number of frames to keep "off"

    // internal member
    int counter;       // counter for state transitions
    int state;        // state of light, -1 off, 1 on, 0 dead
    } BLINKER, *BLINKER_PTR;

// a 2D vertex
typedef struct VERTEX2DI_TYP
    {
    int x,y; // the vertex
    } VERTEX2DI, *VERTEX2DI_PTR;

// a 2D vertex
typedef struct VERTEX2DF_TYP
    {
    float x,y; // the vertex
    } VERTEX2DF, *VERTEX2DF_PTR;

// a 2D polygon
typedef struct POLYGON2D_TYP
    {
    int state;    // state of polygon
    int num_verts;  // number of vertices
```

Playrix Ex. 1023, page 170

05 0672318350 CH03 05•02•2003 2:34 PM Page 146


```
    int x0,y0;    // position of center of polygon
    int xv,yv;    // initial velocity
    DWORD color;   // could be index or PALETTENTRY
    VERTEX2DF *vlist; // pointer to vertex list

    } POLYGON2D, *POLYGON2D_PTR;



// matrix defines
typedef struct MATRIX3X3_TYP
    {
    union
    {
    float M[3][3]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01, M02;
       float M10, M11, M12;
       float M20, M21, M22;
       }; // end explicit names

    }; // end union
    } MATRIX3X3, *MATRIX3X3_PTR;

typedef struct MATRIX1X3_TYP
    {
    union
    {
    float M[3]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01, M02;

       }; // end explicit names
    }; // end union
    } MATRIX1X3, *MATRIX1X3_PTR;

typedef struct MATRIX3X2_TYP
    {
```

Playrix Ex. 1023, page 171

05 0672318350 CH03 05•02•2003 2:34 PM Page 147


```
    union
    {
    float M[3][2]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01;
       float M10, M11;
       float M20, M21;
       }; // end explicit names

    }; // end union
    } MATRIX3X2, *MATRIX3X2_PTR;

typedef struct MATRIX1X2_TYP
    {
    union
    {
    float M[2]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01;

       }; // end explicit names
    }; // end union
    } MATRIX1X2, *MATRIX1X2_PTR;
```


You'll notice some math support at the end. This is from the 2D polygon transformation support from the first *Tricks*. We are going to rewrite much of it when we create the new 3D math library.

Function Prototypes

Next I want you to take a look at the complete function prototype list, so you can see everything at once. I'm not going to cover every single function in the function API in the following sections because many functions are different versions of each other.

```
// DirectDraw functions
int DDraw_Init(int width, int height, int bpp, int windowed=0);
int DDraw_Shutdown(void);
```

Playrix Ex. 1023, page 172

05 0672318350 CH03 05•02•2003 2:34 PM Page 148


```
LPDIRECTDRAWCLIPPER DDraw_Attach_Clipper(LPDIRECTDRAWSURFACE7 lpdds,
                    int num_rects, LPRECT clip_list);

LPDIRECTDRAWSURFACE7 DDraw_Create_Surface(int width, int height,
                int mem_flags=0, USHORT color_key_value=0);

int DDraw_Flip(void);
int DDraw_Wait_For_Vsync(void);

int DDraw_Fill_Surface(LPDIRECTDRAWSURFACE7 lpdds, USHORT color,
            RECT *client=NULL);

UCHAR *DDraw_Lock_Surface(LPDIRECTDRAWSURFACE7 lpdds,int *lpitch);
int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE7 lpdds);
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);
UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);

// BOB functions
int Create_BOB(BOB_PTR bob,int x, int y,int width, int height,
        int num_frames,int attr,
        int mem_flags=0, USHORT color_key_value=0, int bpp=8);

int Clone_BOB(BOB_PTR source, BOB_PTR dest);
int Destroy_BOB(BOB_PTR bob);
int Draw_BOB(BOB_PTR bob, LPDIRECTDRAWSURFACE7 dest);

int Draw_Scaled_BOB(BOB_PTR bob, int swidth, int sheight,
          LPDIRECTDRAWSURFACE7 dest);

int Draw_BOB16(BOB_PTR bob, LPDIRECTDRAWSURFACE7 dest);

int Draw_Scaled_BOB16(BOB_PTR bob, int swidth, int sheight,
           LPDIRECTDRAWSURFACE7 dest);

int Load_Frame_BOB(BOB_PTR bob, BITMAP_FILE_PTR bitmap,
          int frame, int cx,int cy,int mode);

int Load_Frame_BOB16(BOB_PTR bob, BITMAP_FILE_PTR bitmap,
           int frame, int cx,int cy,int mode);
```

Playrix Ex. 1023, page 173

05 0672318350 CH03 05•02•2003 2:34 PM Page 149


```
int Animate_BOB(BOB_PTR bob);
int Move_BOB(BOB_PTR bob);

int Load_Animation_BOB(BOB_PTR bob, int anim_index,
           int num_frames, int *sequence);

int Set_Pos_BOB(BOB_PTR bob, int x, int y);
int Set_Vel_BOB(BOB_PTR bob,int xv, int yv);
int Set_Anim_Speed_BOB(BOB_PTR bob,int speed);
int Set_Animation_BOB(BOB_PTR bob, int anim_index);
int Hide_BOB(BOB_PTR bob);
int Show_BOB(BOB_PTR bob);
int Collision_BOBS(BOB_PTR bob1, BOB_PTR bob2);

// general utility functions
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);

int Collision_Test(int x1, int y1, int w1, int h1,
          int x2, int y2, int w2, int h2);

int Color_Scan(int x1, int y1, int x2, int y2,
        UCHAR scan_start, UCHAR scan_end,
        UCHAR *scan_buffer, int scan_lpitch);

int Color_Scan16(int x1, int y1, int x2, int y2,
         USHORT scan_start, USHORT scan_end,
         UCHAR *scan_buffer, int scan_lpitch);

// graphics functions
int Draw_Clip_Line(int x0,int y0, int x1, int y1, int color,
          UCHAR *dest_buffer, int lpitch);

int Draw_Clip_Line16(int x0,int y0, int x1, int y1, int color,
           UCHAR *dest_buffer, int lpitch);

int Clip_Line(int &x1,int &y1,int &x2, int &y2);

int Draw_Line(int xo, int yo, int x1,int y1, int color,
       UCHAR *vb_start,int lpitch);
```

Playrix Ex. 1023, page 174

05 0672318350 CH03 05•02•2003 2:34 PM Page 150


```
int Draw_Line16(int xo, int yo, int x1,int y1, int color,
        UCHAR *vb_start,int lpitch);

int Draw_Pixel(int x, int y,int color,
        UCHAR *video_buffer, int lpitch);

int Draw_Rectangle(int x1, int y1, int x2, int y2,
          int color,LPDIRECTDRAWSURFACE7 lpdds);

void HLine(int x1,int x2,int y,int color, UCHAR *vbuffer, int lpitch);
void VLine(int y1,int y2,int x,int color, UCHAR *vbuffer, int lpitch);
void HLine16(int x1,int x2,int y,int color, UCHAR *vbuffer, int lpitch);
void VLine16(int y1,int y2,int x,int color, UCHAR *vbuffer, int lpitch);
void Screen_Transitions(int effect, UCHAR *vbuffer, int lpitch);
int Draw_Pixel(int x, int y,int color,UCHAR *video_buffer, int lpitch);
int Draw_Pixel16(int x, int y,int color,UCHAR *video_buffer, int lpitch);

// palette functions
int Set_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette_To_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
int Rotate_Colors(int start_index, int end_index);
int Blink_Colors(int command, BLINKER_PTR new_light, int id);

// simple bitmap image functions
int Create_Bitmap(BITMAP_IMAGE_PTR image, int x, int y,
         int width, int height, int bpp=8);

int Destroy_Bitmap(BITMAP_IMAGE_PTR image);

int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap,UCHAR *dest_buffer,
        int lpitch, int transparent);

int Draw_Bitmap16(BITMAP_IMAGE_PTR source_bitmap,UCHAR *dest_buffer,
         int lpitch, int transparent);

int Load_Image_Bitmap(BITMAP_IMAGE_PTR image,BITMAP_FILE_PTR bitmap,
           int cx,int cy,int mode);
```

Playrix Ex. 1023, page 175

05 0672318350 CH03 05•02•2003 2:34 PM Page 151


```
int Load_Image_Bitmap16(BITMAP_IMAGE_PTR image,BITMAP_FILE_PTR bitmap,
            int cx,int cy,int mode);

int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);

int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap, int dest_x, int dest_y,
        BITMAP_IMAGE_PTR source_bitmap, int source_x, int source_y,
        int width, int height);

int Flip_Bitmap(UCHAR *image, int bytes_per_line, int height);

// bitmap file functions
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, char *filename);
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap);

// gdi functions
int Draw_Text_GDI(char *text, int x,int y,
         COLORREF color, LPDIRECTDRAWSURFACE7 lpdds);

int Draw_Text_GDI(char *text, int x,int y,
         int color, LPDIRECTDRAWSURFACE7 lpdds);

// error functions
int Open_Error_File(char *filename, FILE *fp_override=NULL);
int Close_Error_File(void);
int Write_Error(char *string, ...);

// 2d 8-bit, 16-bit triangle rendering
void Draw_Top_Tri(int x1,int y1,int x2,int y2, int x3,int y3,
         int color,UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_Tri(int x1,int y1, int x2,int y2, int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

void Draw_Top_Tri16(int x1,int y1,int x2,int y2, int x3,int y3,
          int color,UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_Tri16(int x1,int y1, int x2,int y2, int x3,int y3,
            int color,UCHAR *dest_buffer, int mempitch);

void Draw_Top_TriFP(int x1,int y1,int x2,int y2, int x3,int y3,
          int color,UCHAR *dest_buffer, int mempitch);
```


Playrix Ex. 1023, page 176

05 0672318350 CH03 05•02•2003 2:34 PM Page 152


```
void Draw_Bottom_TriFP(int x1,int y1, int x2,int y2, int x3,int y3,
            int color,UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D(int x1,int y1,int x2,int y2,int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D16(int x1,int y1,int x2,int y2,int x3,int y3,
            int color,UCHAR *dest_buffer, int mempitch);

void Draw_TriangleFP_2D(int x1,int y1,int x2,int y2,int x3,int y3,
            int color,UCHAR *dest_buffer, int mempitch);

inline void Draw_QuadFP_2D(int x0,int y0,int x1,int y1,
              int x2,int y2,int x3, int y3,
              int color,UCHAR *dest_buffer, int mempitch);


// general 2D 8-bit, 16-bit polygon rendering and transforming functions
void Draw_Filled_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer, int mempitch);
void Draw_Filled_Polygon2D16(POLYGON2D_PTR poly, UCHAR *vbuffer, int mempitch);
int Translate_Polygon2D(POLYGON2D_PTR poly, int dx, int dy);
int Rotate_Polygon2D(POLYGON2D_PTR poly, int theta);
int Scale_Polygon2D(POLYGON2D_PTR poly, float sx, float sy);
void Build_Sin_Cos_Tables(void);
int Translate_Polygon2D_Mat(POLYGON2D_PTR poly, int dx, int dy);
int Rotate_Polygon2D_Mat(POLYGON2D_PTR poly, int theta);
int Scale_Polygon2D_Mat(POLYGON2D_PTR poly, float sx, float sy);
int Draw_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer, int lpitch);
int Draw_Polygon2D16(POLYGON2D_PTR poly, UCHAR *vbuffer, int lpitch);

// math functions
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

// collision detection functions
int Find_Bounding_Box_Poly2D(POLYGON2D_PTR poly,
              float &min_x, float &max_x,
              float &min_y, float &max_y);

int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod);
```

Playrix Ex. 1023, page 177

05 0672318350 CH03 05•02•2003 2:34 PM Page 153


```
int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma,
          MATRIX3X3_PTR mb,
          MATRIX1X3_PTR mprod);

int Mat_Mul_3X3(MATRIX3X3_PTR ma,
        MATRIX3X3_PTR mb,
        MATRIX3X3_PTR mprod);

inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
            float m00, float m01,
            float m10, float m11,
            float m20, float m21);


// memory manipulation functions
inline void Mem_Set_WORD(void *dest, USHORT data, int count);
inline void Mem_Set_QUAD(void *dest, UINT  data, int count);
```


You'll notice that all the calls to any rasterization functions that take pointers to the frame buffer always take a `UCHAR*`. Additionally, all memory pitch parameters are always in terms of bytes. Remember, this was our convention. Also note that about 90% of the entire library supports 16-bit graphics, but some things only work for 8-bit color. Finally, there's a lot of 2D polygon functions in there. These are all from the first *Tricks*, and although we could just use them for this book, we are going to write all of them again from scratch. This is because they are part of the rasterization process in 3D graphics, and I'm going to have to think up new names for them!

Global Domination

You know that I like globals because they are so fast. Moreover, they are appropriate for a lot of system-level variables (which any game engine has a lot of), so here are the globals for the engine—again, I know this is hard to follow, but just try and get a feel for them. I have commented them to help out.

```
FILE *fp_error;           // general error file
char error_filename[80];       // error file name

// notice that interface 7.0 is used on a number of interfaces
LPDIRECTDRAW7    lpdd;      // dd object
LPDIRECTDRAWSURFACE7 lpddsprimary;  // dd primary surface
LPDIRECTDRAWSURFACE7 lpddsback;   // dd back surface
LPDIRECTDRAWPALETTE lpddpal;    // a pointer to the created dd palette
LPDIRECTDRAWCLIPPER lpddclipper;  // dd clipper for back surface
LPDIRECTDRAWCLIPPER lpddclipperwin; // dd clipper for window
```

Playrix Ex. 1023, page 178

05 0672318350 CH03 05•02•2003 2:34 PM Page 154


```
PALETTEENTRY     palette[256];  // color palette
PALETTEENTRY     save_palette[256]; // used to save palettes
DDSURFACEDESC2    ddsd;   // a direct draw surface description struct
DDBLTFX       ddbltfx; // used to fill
DDSCAPS2       ddscaps; // a direct draw surface capabilities struct
HRESULT       ddrval;  // result back from dd calls
UCHAR        *primary_buffer; // primary video buffer
UCHAR        *back_buffer;  // secondary back buffer
int         primary_lpitch; // memory line pitch
int         back_lpitch;   // memory line pitch
BITMAP_FILE     bitmap8bit;   // a 8 bit bitmap file
BITMAP_FILE     bitmap16bit;   // a 16 bit bitmap file
BITMAP_FILE     bitmap24bit;   // a 24 bit bitmap file

DWORD start_clock_count;  // used for timing
int  windowed_mode;    // tracks if dd is windowed or not

// these defined the general clipping rectangle for software clipping
int min_clip_x, // clipping rectangle
  max_clip_x,
  min_clip_y,
  max_clip_y;

// these are overwritten globally by DD_Init()
int screen_width,  // width of screen
  screen_height,  // height of screen
  screen_bpp,   // bits per pixel
  screen_windowed; // is this a windowed app?

int dd_pixel_format; // default pixel format

int window_client_x0; // used to track the starting (x,y) client area for
int window_client_y0; // for windowed mode directdraw operations

// storage for our lookup tables
float cos_look[360];
float sin_look[360];

// function ptr to RGB16 builder that builds 5.5.5 or 5.6.5 data
// depending on mode
USHORT (*RGB16Bit)(int r, int g, int b);
```

Make note of all the global names of the DirectX interface. Also, take a look at the various clipping and screen size globals.

Playrix Ex. 1023, page 179

05 0672318350 CH03 05•02•2003 2:34 PM Page 155

The DirectDraw Interface

Now that you've seen all the data support, let's take a look at all the DirectDraw support functions. The DirectDraw system from the first *Tricks* (with a little upgrading) has the following features:

- Double-buffered, with both a primary and secondary (offscreen frame buffer)
- 8-bit color with palette
- 16-bit color with auto pixel format detection
- Windowed support for both 8- and 16-bit color modes
- 2D polygon and raster bitmap clipping
- Access to both the primary and secondary buffers
- Page flipping and buffer copying for windowed displays

NOTE

Note the last feature. DirectX does *not* allow Windowed applications to be double buffered, or in DirectX speak, to be *complex surfaces*. Therefore, when a windowed display is requested, instead of creating a standard full-screen complex surface with a primary buffer and a back buffer, only a primary buffer (which is the entire screen), and an offscreen plain buffer that's the size of the client area of the window are created. Then when a flip is requested, the logic manually copies the offscreen back buffer to the window's client area. Of course, this is all transparent to you.

Now let's take a look at each function, along with an example.

Function Prototype:

```
int DDraw_Init(int width, // width of display
        int height, // height of display
        int bpp,  // bits per pixel
        int windowed=0); // 0 for full screen, 1 for windowed
```

Purpose:

`DDraw_Init()` starts up and initializes DirectDraw. You can send any resolution and color depth. Also, if you want a windowed display, send a `1` for the last parameter; otherwise it defaults to full-screen. Returns `TRUE` if successful.

Examples:

```
// put the system into full screen 800x600 with 256 colors
DDraw_Init(800,600,8);
```

Playrix Ex. 1023, page 180

05 0672318350 CH03 05•02•2003 2:34 PM Page 156


```
// put the system into windowed mode screen 400x400 with 16-bit color
// note that the desktop must be in 16-bit color
DDraw_Init(400,400,16,1);
```

Function Prototype:

```
int DDraw_Shutdown(void);
```

Purpose:

`DDraw_Shutdown()` shuts down DirectDraw and releases all interfaces.

Example:

```
// in your system shutdown code you might put
DDraw_Shutdown();
```

Function Prototype:

```
LPDIRECTDRAWCLIPPER
 DDraw_Attach_Clipper(
  LPDIRECTDRAWSURFACE7 lpdds, // surface to attach to
  int num_rects,   // number of rects
  LPRECT clip_list); // pointer to rects
```

Purpose:

`DDraw_Attach_Clipper()` attaches a clipper to the sent surface (the back buffer in most cases). In addition, you must send the number of rectangles in the clipping list and a pointer to the `RECT` list itself. Returns `TRUE` if successful.

Example:

```
// creates a clipping region the size of the screen
RECT clip_zone = {0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1};
DDraw_Attach_Clipper(lpddsback, 1, &clip_zone);
```

Function Prototype:

```
LPDIRECTDRAWSURFACE7
  DDraw_Create_Surface(int width, // width of surface
       int height,     // height of surface
       int mem_flags=0,   // control flags
       USHORT color_key_value=0); // color key value
```

Purpose:

`DDraw_Create_Surface()` is used to create a generic offscreen DirectDraw surface in system memory, VRAM, or AGP memory. The default is `DDSCAPS_OFFSCREENPLAIN`. Any additional

Playrix Ex. 1023, page 181

05 0672318350 CH03 05•02•2003 2:34 PM Page 157

control flags are logically ORed with the default. They are the standard DirectDraw DDSCAP* flags, such as DDSCAPS_SYSTEMMEMORY and DDSCAPS_VIDEOMEMORY for system memory and VRAM, respectively. The function has internal logic to create either an 8- or 16-bit surface based on the previously set graphics mode. Additionally, the last parameter controls the value of the color key, which defaults to (int)0. In 8-bit color mode, this equates to index 0; in 16-bit color mode, this equates to RGB 0.0.0. If you want another value to be used as the transparent color key, send the value to override the default. If the function is successful, it returns a pointer to the new surface; otherwise it returns NULL.

Examples:

```
// let's create a 64x64 surface in VRAM
LPDIRECTDRAWSURFACE7 image = DDraw_Create_Surface(64,64, DDSCAPS_VIDEOMEMORY);

// let's create a 128x128 surface in off screen memory with a color key of 16
// assuming this is a 256 color mode
LPDIRECTDRAWSURFACE7 image =
       DDraw_Create_Surface(128,128, DDSCAPS_OFFSCREENPLAIN ,16);
```

Function Prototype:

```
int DDraw_Flip(void);
```

Purpose:

DDraw_Flip() simply flips the primary surface with the secondary surface in full-screen mode; in windowed mode, it copies the virtual secondary or back buffer to the client area of the windowed display. The call waits until the flip can take place, so it might not return immediately. Returns TRUE if successful.

Example:

```
// flip em baby
DDraw_Flip();
```

Function Prototype:

```
int DDraw_Wait_For_Vsync(void);
```

Purpose:

DDraw_Wait_For_Vsync() waits until the next vertical blank period begins (when the raster hits the bottom of the screen). Returns TRUE if successful, FALSE if something really bad happened.

Example:

```
// wait 1/70th of sec
DDraw_Wait_For_Vsync();
```

Playrix Ex. 1023, page 182

05 0672318350 CH03 05•02•2003 2:34 PM Page 158

Function Prototype:

```
int DDraw_Fill_Surface(LPDIRECTDRAWSURFACE7 lpdds, // surface to fill
            int color, // color to fill
                 // either 8-bit index or 16-bit RGB
            RECT *client); // client area, null for whole surface
```

Purpose:

`DDraw_Fill_Surface()` is used to fill a surface with a color. The color must be in the color depth format of the surface. For example, a single byte in 256-color mode or an RGB descriptor in high-color modes. Also, if you want to fill a subregion of the entire surface, you can send a `RECT*` that defines it; otherwise, the entire surface is filled. Returns `TRUE` if successful.

Example:

```
// fill the primary surface with color 0
DDraw_Fill_Surface(lpddsprimary,0);
```

Function Prototype:

```
UCHAR *DDraw_Lock_Surface(LPDIRECTDRAWSURFACE7 lpdds,int *lpitch);
```

Purpose:

`DDraw_Lock_Surface()` locks the sent surface (if possible) and returns a `UCHAR` pointer to the surface, along with updating the sent `lpitch` variable with the linear memory pitch of the surface. While the surface is locked, you can manipulate it and write pixels to it, but the blitter will be blocked, so remember to unlock the surface ASAP. In addition, after unlocking the surface, the memory pointer and pitch are most likely invalid and should not be used. Returns the non-`NULL` address of the surface memory if successful, `NULL` otherwise.

Example:

```
// holds the memory pitch
int lpitch = 0;

// let's lock the little 64x64 image we made
UCHAR *memory = DDraw_Lock_Surface(image, &lpitch);
```

Function Prototype:

```
int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE7 lpdds);
```

Playrix Ex. 1023, page 183

05 0672318350 CH03 05•02•2003 2:34 PM Page 159

Purpose:

`DDraw_Unlock_Surface()` unlocks a surface previously locked with `DDraw_Lock_Surface()`. You need only send the pointer to the surface. Returns `TRUE` if successful.

Example:

```
// unlock the image surface
DDraw_Unlock_Surface(image);
```

Function Prototype(s):

```
UCHAR *DDraw_Lock_Back_Surface(void);
UCHAR *DDraw_Lock_Primary_Surface(void);
```

Purpose:

These two functions are used to lock the primary and secondary rendering surfaces. However, in most cases you'll only be interested in locking the secondary surface, because this is a double-buffered system, but the capability to lock the primary surface is there if you need it. If you call `DDraw_Lock_Primary_Surface()`, the following globals will become valid:

```
extern UCHAR *primary_buffer;   // primary video buffer
extern int   primary_lpitch;    // memory line pitch
```

Then you are free to manipulate the surface memory as you want; however, the blitter will be blocked and all hardware acceleration is inhibited during a lock. Anyway, making the call to `DDraw_Lock_Back_Surface()` will lock the back buffer surface and validate the following globals:

```
extern UCHAR *back_buffer;     // secondary back buffer
extern int   back_lpitch;     // memory line pitch
```

> **NOTE**
>
> Do not change any of these globals yourself—they are used to track state changes in the locking functions. Changing them yourself might make the engine go crazy.

Example:

```
// let lock the primary surface and write a pixel to the
// upper left hand corner assuming an 8-bit mode
DDraw_Lock_Primary();

primary_buffer[0] = 100;
```

Playrix Ex. 1023, page 184

05 0672318350 CH03 05•02•2003 2:34 PM Page 160

Function Prototype:

```
int DDraw_Unlock_Primary_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

Purpose:

These functions are used to unlock the primary or back buffer surfaces. If you try to unlock a surface that wasn't locked, there is no effect. Returns `TRUE` if successful.

Example:

```
// unlock the secondary back buffer
DDraw_Unlock_Back_Surface();
```

2D Polygon Functions

The next set of functions make up the 2D polygon system. This is by no means advanced, fast, or cutting edge, but it's great to build up demos. The functions work and do the job. However, there are better ways to do all of this stuff, but that's why you're glued to the book, right?

Function Prototype(s):

```
void Draw_Triangle_2D(int x1,int y1, // triangle vertices
         int x2,int y2,
         int x3,int y3,
         int color, // 8-bit color index
         UCHAR *dest_buffer, // destination buffer
         int mempitch); // memory pitch

// fixed point high speed version, slightly less accurate
void Draw_TriangleFP_2D(int x1,int y1,
            int x2,int y2,
            int x3,int y3,
            int color,
            UCHAR *dest_buffer,
            int mempitch);

// 16-bit version
void Draw_Triangle_2D16(int x1,int y1, // triangle vertices
         int x2,int y2,
         int x3,int y3,
         int color, // 16-bit RGB color
         UCHAR *dest_buffer, // destination buffer
         int mempitch); // memory pitch
```

Playrix Ex. 1023, page 185

05 0672318350 CH03 05•02•2003 2:34 PM Page 161

Purpose:

`Draw_Triangle_2D*()` draws a filled triangle in the given memory buffer with the sent color. The triangle will be clipped to the current clipping region set in the globals, *not* by the DirectDraw clipper, because the function uses software and not the blitter to draw lines. `Draw_TriangleFP_2D()` does the exact same thing, but uses fixed-point math internally and is slightly faster, but with slightly less accuracy. All functions return nothing.

Examples:

```
// draw a triangle (100,10) (150,50) (50,60)
// with color index 50 in the back buffer surface
Draw_Triangle_2D(100,10,150,50,50,60,
         50, // color index 50
         back_buffer,
         back_lpitch);

// do the same with 16 bit version
// with RGB color pure red
Draw_Triangle_2D16(100,10,150,50,50,60,
         RGB16BIT565(31,0,0), // assume 5.6.5 format
         back_buffer,
         back_lpitch);
```

Function Prototype:

```
inline void Draw_QuadFP_2D(int x0,int y0, // vertices
      int x1,int y1,
      int x2,int y2,
      int x3,int y3,
      int color, // 8-bit color index
      UCHAR *dest_buffer, // destination video buffer
      int mempitch); // memory pitch of buffer
```

Purpose:

`Draw_QuadFP_2D()` draws the sent quadrilateral as a composition of two triangles in 8-bit mode only. Notice there isn't a 16-bit version. Returns nothing.

Example:

```
// draw a quadrilateral, note vertices must be ordered
// either in cw or ccw order
Draw_QuadFP_2D(0,0, 10,0, 15,20, 5,25,
        100,
        back_buffer, back_lpitch);
```

Playrix Ex. 1023, page 186

05 0672318350 CH03 05•02•2003 2:34 PM Page 162

Function Prototype:

```
void Draw_Filled_Polygon2D(
     POLYGON2D_PTR poly, // poly to render
     UCHAR *vbuffer, // video buffer
     int mempitch); // memory pitch

// 16-bit version
void Draw_Filled_Polygon2D16(
     POLYGON2D_PTR poly, // poly to render
     UCHAR *vbuffer, // video buffer
     int mempitch); // memory pitch
```

Purpose:

`Draw_Filled_Polygon2D*()` draws a general filled polygon with *n* sides in either 8- or 16-bit color mode. The function simply takes the polygon to render, a pointer to the video buffer along with pitch, and that's it! Note that the function renders relative to the polys `(x0,y0)`, so make sure these are initialized. Returns nothing.

Examples:

```
// draw a polygon in the primary buffer
// in 8-bit mode
Draw_Filled_Polygon2D(&poly,
           primary_buffer,
           primary_lpitch);

// draw a polygon in the primary buffer
// in 16-bit mode
Draw_Filled_Polygon2D16(&poly,
           primary_buffer,
           primary_lpitch);
```

Function Prototype:

```
int Translate_Polygon2D(
      POLYGON2D_PTR poly, // poly to translate
      int dx, int dy); // translation factors
```

Purpose:

`Translate_Polygon2D()` translates the given polygon's origin `(x0,y0)`. Note that the function does not transform or modify the actual vertices making up the polygon. Returns `TRUE` if successful.

Playrix Ex. 1023, page 187

05 0672318350 CH03 05•02•2003 2:34 PM Page 163

Example:

```
// translate polygon 10,-5
Translate_Polygon2D(&poly, 10, -5);
```

Function Prototype:

```
int Rotate_Polygon2D(
         POLYGON2D_PTR poly, // poly to rotate
         int theta); // angle 0-359
```

Purpose:

`Rotate_Polygon2D()` rotates the sent polygon in a counter-clockwise fashion about its origin. The angle must be an integer from 0–359. Returns `TRUE` if successful.

Example:

```
// rotate polygon 10 degrees
Rotate_Polygon2D(&poly, 10);
```

Function Prototype:

```
int Scale_Polygon2D(POLYGON2D_PTR poly, // poly to scale
          float sx, float sy); // scale factors
```

Purpose:

`Scale_Polygon2D()` scales the sent polygon by scale factors `sx` and `sy` in the x- and y-axes, respectively. Returns nothing.

Example:

```
// scale the poly equally 2x
Scale_Polygon2D(&poly, 2,2);
```

2D Graphic Primitives

This set of functions contains a few of everything—kind of a potpourri of graphics primitives. Nothing you haven't seen, at least I don't think so. Also, the 16-bit support is there, but not completely orthogonal to the 8-bit support.

Function Prototype:

```
int Draw_Clip_Line(int x0,int y0, // starting point
          int x1, int y1, // ending point
          int color,   // 8-bit color
          UCHAR *dest_buffer, // video buffer
          int lpitch); // memory pitch
```

Playrix Ex. 1023, page 188

05 0672318350 CH03 05•02•2003 2:34 PM Page 164


```
// 16-bit version
int Draw_Clip_Line16(int x0,int y0, // starting point
          int x1, int y1, // ending point
          int color,   // 16-bit RGB color
          UCHAR *dest_buffer, // video buffer
          int lpitch); // memory pitch
```

Purpose:

`Draw_Clip_Line*()` clips the sent line to the current clipping rectangle and then draws a line in the sent buffer (8- or 16-bit). Returns `TRUE` if successful.

Examples:

```
// draw a line in the back buffer from (10,10) to (100,200)
//assume an 8-bit system
Draw_Clip_Line(10,10,100,200,
        5, // color index 5
        back_buffer,
        back_lpitch);

// draw a line in the back buffer from (10,10) to (100,200)
//assume an 16-bit system
Draw_Clip_Line16(10,10,100,200,
        RGB16BIT565(0,31,0), // RGB green
        back_buffer,
        back_lpitch);
```

Function Prototype:

```
int Clip_Line(int &x1,int &y1,  // starting point
       int &x2, int &y2); // ending point
```

Purpose:

`Clip_Line()` is for the most part internal, but you can call it to clip the sent line to the current clipping rectangle. Note that the function modifies the sent endpoints, so save them if you don't want this side effect. Also, the function does NOT draw anything, it only clips the endpoints mathematically, so bit depth is irrelevant. Returns `TRUE` if successful.

Example:

```
// clip the line defined by x1,y1 to x2,y2
Clip_Line(x1,y1,x2,y2);
```

Playrix Ex. 1023, page 189

05 0672318350 CH03 05•02•2003 2:34 PM Page 165

Function Prototype:

```
int Draw_Line(int xo, int yo, // starting point
       int x1,int y1,  // ending point
       int color,    // 8-bit color index
       UCHAR *vb_start, // video buffer
       int lpitch);   // memory pitch


// 16-bit version
int Draw_Line16(int xo, int yo, // starting point
       int x1,int y1,  // ending point
       int color,    // 16-bit RGB color
       UCHAR *vb_start, // video buffer
       int lpitch);   // memory pitch
```


Purpose:

`Draw_Line*()` draws a line without any clipping, so make sure that the endpoints are within the display surface's valid coordinates. This function is slightly faster than the clipped version, because the clipping operation is not needed. Returns `TRUE` if successful.

Example:

```
// draw a line in the back buffer from (10,10) to (100,200)
// 8-bit mode
Draw_Line(10,10,100,200,
     5, // color 5
     back_buffer,
     back_lpitch);

// draw a line in the back buffer from (10,10) to (100,200)
// 16-bit mode
Draw_Line16(10,10,100,200,
      RGB16BIT(31,31,31), // RGB white
      back_buffer,
      back_lpitch);
```

Function Prototype:

```
inline int Draw_Pixel(int x, int y, // position of pixel
           int color, // 8-bit color
           UCHAR *video_buffer, // gee hmm?
           int lpitch); // memory pitch
```

Playrix Ex. 1023, page 190

05 0672318350 CH03 05•02•2003 2:34 PM Page 166


```
// 16-bit version
inline int Draw_Pixel16(int x, int y, // position of pixel
            int color, // 16-bit RGB color
            UCHAR *video_buffer, // gee hmm?
            int lpitch); // memory pitch
```

Purpose:

`Draw_Pixel*()` draws a single pixel on the sent display surface memory buffer. In most cases, you won't create objects based on pixels, because the overhead of the call itself takes more time than plotting the pixel. If speed isn't your concern, the function does the job—at least it's inline! Returns `TRUE` if successful.

Example:

```
// draw pixel with color index 100 in the center of the 640x480x8 screen
Draw_Pixel(320,240, 100, back_buffer, back_lpitch);

// draw blue pixel in the center of the 640x480x16 screen
Draw_Pixel16(320,240, RGB16BIT(0,0,31), back_buffer, back_lpitch);
```

Function Prototype:

```
int Draw_Rectangle(int x1, int y1, // upper left corner
        int x2, int y2, // lower right corner
        int color, // 8-bit color index or
              // 16-bit RGB word
        LPDIRECTDRAWSURFACE7 lpdds); // dd surface
```

Purpose:

`Draw_Rectangle()` draws a rectangle on the sent DirectDraw surface. Note that the surface must be unlocked for the call to work. Moreover, the function uses the blitter, so it's very fast, and it works in either 8- or 16-bit color mode. Returns `TRUE` if successful.

Example:

```
// fill the screen using the blitter
Draw_Rectangle(0,0,639,479,0,lpddsback);
```

Function Prototype:

```
void HLine(int x1,int x2, // start and end x points
      int y,     // row to draw on
      int color,   // 8-bit color
      UCHAR *vbuffer, // video buffer
      int lpitch); // memory pitch
```

Playrix Ex. 1023, page 191

05 0672318350 CH03 05•02•2003 2:34 PM Page 167


```
void HLine16(int x1,int x2, // start and end x points
      int y,     // row to draw on
      int color,   // 16-bit RGB color
      UCHAR *vbuffer, // video buffer
      int lpitch); // memory pitch
```

Purpose:

`HLine*()` draws a horizontal line very quickly compared to the general line drawing function. Returns nothing.

Example:

```
// draw a fast line in 8-bit mode from 10,100 to 100,100 with color index 20
HLine(10,100,100,
   20, back_buffer, back_lpitch);

// draw a fast line in 16-bit mode from 10,100 to 100,100 with blue
HLine(10,100,100,
   RGB16Bit(0,0,255), back_buffer, back_lpitch);
```

Function Prototype:

```
void VLine(int y1,int y2, // start and end row
      int x,     // column to draw in
      int color,   // 8-bit color
      UCHAR *vbuffer,// video buffer
      int lpitch);  // memory pitch

void VLine16(int y1,int y2, // start and end row
      int x,     // column to draw in
      int color,   // 16-bit RGB color
      UCHAR *vbuffer,// video buffer
      int lpitch);  // memory pitch
```

Purpose:

`VLine()` draws a fast vertical line. It's not as fast as `HLine()`, but it is faster than `Draw_Line()`, so use it if you know a line is going to be vertical in all cases. Returns nothing.

Example:

```
// draw a 8-bit line from 320,0 to 320,479 with color index 54
VLine(0,479,320,54,
   primary_buffer,
   primary_lpitch);
```

Playrix Ex. 1023, page 192

05 0672318350 CH03 05•02•2003 2:34 PM Page 168


```
// draw a 16-bit line from 320,0 to 320,479 with green
VLine(0,479,320,RGB16Bit(0,255,0),
   primary_buffer,
   primary_lpitch);
```

Function Prototype:

```
void Screen_Transitions(int effect,  // screen transition
            UCHAR *vbuffer,// video buffer
            int lpitch);  // memory pitch

// screen transition commands
#define SCREEN_DARKNESS 0     // fade to black
#define SCREEN_WHITENESS 1     // fade to white
#define SCREEN_SWIPE_X  2     // do a horizontal swipe
#define SCREEN_SWIPE_Y  3     // do a vertical swipe
#define SCREEN_DISOLVE  4     // a pixel disolve
#define SCREEN_SCRUNCH  5     // a square compression
#define SCREEN_BLUENESS 6     // fade to blue
#define SCREEN_REDNESS  7     // fade to red
#define SCREEN_GREENNESS 8     // fade to green
```

Purpose:

`Screen_Transition()` performs various in-memory screen transitions as listed in the preceding header information. Note that the transformations are destructive, so please save the image and/or palette if you need them after the transition. Note that this only works for 8-bit color modes. Returns nothing.

Example:

```
// fade the primary display screen to black
Screen_Transition(SCREEN_DARKNESS, NULL, 0);
```

Function Prototype(s):

```
int Draw_Text_GDI(char *text, // null terminated string
      int x,int y, // position
      COLORREF color, // general RGB color
      LPDIRECTDRAWSURFACE7 lpdds); // dd surface

int Draw_Text_GDI(char *text, // null terminated string
        int x,int y, // position
        int color, // 8-bit color index
        LPDIRECTDRAWSURFACE7 lpdds); // dd surface
```

Playrix Ex. 1023, page 193

05 0672318350 CH03 05•02•2003 2:34 PM Page 169

Purpose:

`Draw_Text_GDI()` draws GDI text on the sent surface with the desired color and position in either 8- or 16-bit color modes. The function is overloaded to take both a `COLORREF` in the form of the `RGB()` macro or a 256 (8-bit) color index. Note that the destination surface must be unlocked for the function to operate, because it locks it momentarily to perform the text blitting with GDI. Returns `TRUE` if successful.

Example:

```
// draw text with color RGB(100,100,0);
Draw_Text_GDI("This is a test",100,50,
       RGB16Bit(100,100,0),lpddsprimary);
```


Math and Error Functions

We will rewrite the math library quite a bit, but we might as well take a look at what there is to offer, because I might use it in some of the early demos. Additionally, there is some error support that you can use to write errors or diagnostic text as the engine runs. This is helpful, because debugging DirectX applications is so hard because of their resource sharing, or lack thereof <GRIN>.

Function Prototype:

```
int Fast_Distance_2D(int x, int y);
```

Purpose:

`Fast_Distance()` computes the distance from (0,0) to (x,y) using a fast approximation. Returns the distance within a 3.5% error truncated to an integer. It uses a Taylor series approximation, if you're interested.

Example:

```
int x1=100,y1=200; // object one
int x2=400,y2=150; // object two

// compute the distance between object one and two
int dist = Fast_Distance_2D(x1-x2, y1-y2);
```

Function Prototype:

```
float Fast_Distance_3D(float x, float y, float z);
```

Purpose:

`Fast_Distance_3D()` computes the distance from (0,0,0) to (x,y,z) using a fast approximation. The function returns the distance within an 11% margin of error.

Playrix Ex. 1023, page 194

05 0672318350 CH03 05•02•2003 2:34 PM Page 170

Example:

```
// compute the distance from (0,0,0) to (100,200,300)
float dist = Fast_Distance_3D(100,200,300);
```

Function Prototype:

```
int Find_Bounding_Box_Poly2D(
        POLYGON2D_PTR poly, // the polygon
        float &min_x, float &max_x, // bounding box
        float &min_y, float &max_y);
```

Purpose:

`Find_Bounding_Box_Poly2D()` computes the smallest rectangle that contains the sent polygon in poly. Returns `TRUE` if successful. Also, notice the function takes parameters by reference.

Example:

```
POLYGON2D poly; // assume this is initialized
int min_x, max_x, min_y, max_y; // hold result

// find bounding box
Find_Bounding_Box_Poly2D(&poly,min_x,max_x,min_y,max_y);
```

Function Prototype:

```
int Open_Error_File(char *filename);
```

Purpose:

`Open_Error_File()` opens a disk file that receives error messages sent by you via the `Write_Error()` function. Returns `TRUE` if successful.

Example:

```
// open a general error log
Open_Error_File("errors.log");
```

Function Prototype:

```
int Close_Error_File(void);
```

Purpose:

`Close_Error_File()` closes a previously opened error file. Basically, it shuts down the stream. If you call this and an error file is not open, nothing will happen. Returns `TRUE` if successful.

Playrix Ex. 1023, page 195

05 0672318350 CH03 05•02•2003 2:34 PM Page 171

Example:

```
// close the error system, note no parameter needed
Close_Error_File();
```

Function Prototype:

```
int Write_Error(char *string, ...); // error formatting string
```

Purpose:

`Write_Error()` writes an error out to the previously opened error file. If there is no file open, the function returns a `FALSE` and there is no harm. Note that the function uses the (…) variable parameter indicator, so you can use this function as you would `printf()`. Returns `TRUE` if successful.

Example:

```
// write out some stuff
Write_Error("\nSystem Starting…");
Write_Error("\nx-vel = %d", y-vel = %d", xvel, yvel);
```

Bitmap Functions

The following function set makes up the `BITMAP_IMAGE` and `BITMAP_FILE` manipulation routines. There are functions to load 8-, 16-, 24-, 32-bit bitmaps as well as to extract images from them and create simple `BITMAP_IMAGE` objects (which are not DirectDraw surfaces). In addition, functionality exists to draw these images in 8- and 16-bit modes only, but there is no clipping support. Hence, you can modify the source yourself if you need clipping support, or step up to the *BOB (Blitter Object)*, which is described at the end of the section. We will probably use these functions (or variants) to help load texture and light maps for our 3D rendering, so it's a good idea to take a close look at them.

Function Prototype:

```
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, // bitmap file
           char *filename); // disk .BMP file to load
```

Purpose:

`Load_Bitmap_File()` loads a .BMP format bitmap file off disk into the sent `BITMAP_FILE` structure, where you can manipulate it as you will. The function loads 8-, 16-, and 24-bit bitmaps, as well as the palette information on 8-bit .BMP files. Returns `TRUE` if successful.

Example:

```
// let's load "andre.bmp" off disk
BITMAP_FILE bitmap_file;

Load_Bitmap_File(&bitmap_file, "andre.bmp");
```

Playrix Ex. 1023, page 196

05 0672318350 CH03 05•02•2003 2:34 PM Page 172

Function Prototype:

```
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap); // bitmap to close and unload
```

Purpose:

`Unload_Bitmap_File()` deallocates the memory associated with the image buffer of a loaded `BITMAP_FILE`. Call this function when you have copied the image bits and/or are done working with a particular bitmap. You can reuse the structure, but the memory must be freed first. Returns `TRUE` if successful.

Example:

```
// close the file we just opened
Unload_Bitmap_File(&bitmap_file);
```

Function Prototype:

```
int Create_Bitmap(BITMAP_IMAGE_PTR image, // bitmap image
        int x, int y, // starting position
        int width, int height, // size of bitmap
        int bpp=8); // the bit depth of bitmap
```

Purpose:

`Create_Bitmap()` creates an 8- or 16-bit system memory bitmap at the given position with the given size. The bitmap is initially blank and stored in the `BITMAP_IMAGE` image. The bitmap is not a DirectDraw surface, so there is no acceleration or clipping available. The function defaults to 8-bit bitmaps unless you override the `bpp` parameter with 16 for 16-bit. Returns `TRUE` if successful.

> **NOTE**
>
> There is a big difference between a `BITMAP_FILE` and a `BITMAP_IMAGE`. A `BITMAP_FILE` is a disk .BMP file, whereas a `BITMAP_IMAGE` is a system memory object like a sprite that can be moved and drawn.

Examples:

```
// let's create a 64x64 8-bit bitmap image at (0,0)
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 64,64);

// and now a 16-bit 32x32 image at (100,100)
BITMAP_IMAGE ship2;

// note the addition of 16 to override the default 8-bit depth
Create_Bitmap(&ship, 0,0, 32,32,16);
```

Playrix Ex. 1023, page 197

05 0672318350 CH03 05•02•2003 2:34 PM Page 173

Function Prototype:

```
int Destroy_Bitmap(BITMAP_IMAGE_PTR image); // bitmap image to destroy
```

Purpose:

`Destroy_Bitmap()` is used to release the memory allocated during the creation of a `BITMAP_IMAGE` object (either 8- or 16-bit). You should call this function on your object when you're all done working with it—usually during the shutdown of the game, or if it's been destroyed in a bloody battle. Returns `TRUE` if successful.

Example:

```
// destroy the previously created BITMAP_IMAGE
Destroy_Bitmap(&ship);
```

Function Prototype:

```
int Load_Image_Bitmap(
 BITMAP_IMAGE_PTR image, // bitmap to store image in
 BITMAP_FILE_PTR bitmap, // bitmap file object to load from
 int cx,int cy, // coordinates where to scan (cell or abs)
 int mode); // image scan mode: cell based or absolute

// 16-bit version
int Load_Image_Bitmap16(
 BITMAP_IMAGE_PTR image, // bitmap to store image in
 BITMAP_FILE_PTR bitmap, // bitmap file object to load from
 int cx,int cy, // coordinates where to scan (cell or abs)
 int mode); // image scan mode: cell based or absolute


#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS  1
```

Purpose:

`Load_Image_Bitmap*()` is used to scan an image from a previously loaded `BITMAP_FILE` object into the sent `BITMAP_IMAGE` storage area—this is how you actually get objects and image bits into a `BITMAP_IMAGE`. Of course, the bitmap file and the bitmap object must be the same bit depth (8- or 16-bit), and you have to use the appropriate version of the function.

To use the function, you first must have a `BITMAP_FILE` loaded and have created the `BITMAP_IMAGE`. Then you make the call to scan an image of the same size out of the bitmap data stored in the `BITMAP_FILE`. There are two ways the function works: cell mode or absolute mode.

Playrix Ex. 1023, page 198

05 0672318350 CH03 05•02•2003 2:34 PM Page 174

In cell mode (`BITMAP_EXTRACT_MODE_CELL`), the image is scanned, making the assumption that all the images are in the .BMP file in a template that is some given size *m*×*n*, with a 1 pixel-thick border between each cell, which usually range from 8×8, 16×16, 32×32, 64×64, and so on. Take a look at `TEMPLATE*.BMP` on the CD from this chapter—it contains a number of templates. Cell numbers range from left to right, top to bottom, and start with (0,0).

The second mode of operation is absolute coordinate mode (`BITMAP_EXTRACT_MODE_ABS`). In this mode, the image is scanned at the exact coordinates, sent in `cx`, `cy`. This method is good if you want to load your artwork with various sized images on the same .BMP. Hence, you can't template them.

Example:

```
8-bit example
// assume the source bitmap .BMP file is 640x480x8 and
// has a 8x8 matrix of cells that are each 32x32
// then to load the 3rd cell to the right on the 2nd
// row (cell 2,1), you would do this

// load in the .BMP file into memory
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"images.bmp");


// initialize the bitmap
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 32,32);

// now scan out the data
Load_Image_Bitmap(&ship, &bitmap_file, 2,1,
         BITMAP_EXTRACT_MODE_CELL);

// the exact same example in 16-bit mode
// assume the source bitmap .BMP file is 640x480x16 and
// has a 8x8 matrix of cells that are each 32x32
// then to load the 3rd cell to the right on the 2nd
// row (cell 2,1), you would do this

// load in the .BMP file into memory
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"images.bmp");
```

Playrix Ex. 1023, page 199

05 0672318350 CH03 05•02•2003 2:34 PM Page 175


```
// initialize the bitmap
BITMAP_IMAGE ship2;
Create_Bitmap(&ship2, 0,0, 32,32,16);
************ 0,0,31,31,16? it's ok:al

// now scan out the data
Load_Image_Bitmap16(&ship2, &bitmap_file, 2,1,
         BITMAP_EXTRACT_MODE_CELL);
```

To load the exact same image assuming it's still in the template, but using the absolute mode, we have to figure out the coordinates—remember, there's a 1 pixel-thick partitioning wall on each side of the image.

```
// 8-bit example
Load_Image_Bitmap(&ship, &bitmap_file,
         2*(32+1)+1,1*(32+1)+1,
         BITMAP_EXTRACT_MODE_ABS);
```

Function Prototype:

```
int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap, // bitmap to draw
        UCHAR *dest_buffer, // video buffer
        int lpitch, // memory pitch
        int transparent); // transparency?

// 16-bit version
int Draw_Bitmap16(BITMAP_IMAGE_PTR source_bitmap, // bitmap to draw
        UCHAR *dest_buffer, // video buffer
        int lpitch, // memory pitch
        int transparent); // transparency?
```

Purpose:

`Draw_Bitmap*()` draws the sent bitmap on the destination memory surface in either 8- or 16-bit mode with or without transparency. If transparent is 1, transparency is enabled and any pixel that is color index `0` or RGB word 0.0.0 in 16-bit mode will not be copied. Function returns `TRUE` if successful.

Example:

```
// draw our little ship on the back buffer
// in 8-bit mode
Draw_Bitmap( &ship, back_buffer, back_lpitch, 1);

// and now in 16-bit mode
Draw_Bitmap16( &ship2, back_buffer, back_lpitch, 1);
```

Playrix Ex. 1023, page 200

05 0672318350 CH03 05•02•2003 2:34 PM Page 176

Function Prototype:

```
int Flip_Bitmap(UCHAR *image, // image bits to vertically flip
        int bytes_per_line, // bytes per line
        int height); // total rows or height
```

Purpose:

`Flip_Bitmap()` is usually called internally to flip upside down .BMP files during loading, but you might want to use it to flip an image yourself (8- or 16-bit). The function does an in-memory flip, and actually inverts the bitmap line by line, and your original sent data will be inverted, so watch out! Returns `TRUE` if successful.

Example:

```
// for fun flip the image bits of our little ship
// assuming 1 byte per pixel in 8 bit mode
Flip_Bitmap(ship->buffer, ship->width, ship->height);

// and here's the call for 16-bit mode, same call
// but we need to double the width since there are
// 2 bytes per pixel
Flip_Bitmap(ship2->buffer, 2*ship2->width, ship2->height);
```

Function Prototype:

```
int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap,
        int dest_x, int dest_y,
        BITMAP_IMAGE_PTR source_bitmap,
        int source_x, int source_y,
        int width, int height);
```

Purpose:

`Copy_Bitmap()` copies a rectangular region from one bitmap to another. The source and destination can be the same bitmap; however, the regions must *not* overlap—otherwise, undefined results might occur. Returns `TRUE` if successful.

Example:

```
// load in the .BMP file into memory
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"playfield.bmp");


// initialize a bitmap to hold playfield
BITMAP_IMAGE playfield;
Create_Bitmap(&playfield, 0,0, 400,400,16);
```

Playrix Ex. 1023, page 201

05 0672318350 CH03 05•02•2003 2:34 PM Page 177


```
// now scan out the data
Load_Image_Bitmap16(&playfield, &bitmap_file, 0,0,
         BITMAP_EXTRACT_MODE_ABSOLUTE);


// now copy the top portion of the same bitmap to the bottom
// really chunky scrolling!
Copy_Bitmap(&playfield, 0,200,
      &playfield, 0,0,
      200,200);
```


Function Prototype:

```
int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);
```

Purpose:

`Scroll_Bitmap()` scrolls the sent bitmap horizontally and/or vertically an amount *dx,dy* in the x- and y-axes, respectively. Positive numbers scroll right and down; negative numbers scroll left and up. Returns `TRUE` if successful.

Example:

```
// scroll the playfield 2 pixels to the right
Scroll_Bitmap(&playfield, 2,0);
```

8-Bit Palette Functions

The following functions make up the 8-bit 256 color palette interface. These functions are only relevant if you have the display set for a 256-color mode (8-bit color). However, in windowed display modes, the first and last 10 colors are used by Windows, and you can't alter them. (Of course, in full-screen mode you can.)

Additionally, the basic data structure used to hold a color is a native Win32 structure called `PALETTEENTRY`:

```
typedef struct tagPALETTEENTRY { // pe
  BYTE peRed;  // red channel 8-bits
  BYTE peGreen; // green channel 8-bits
  BYTE peBlue;  // blue channel 8-bits
  BYTE peFlags; // flags control, PC_EXPLICIT for windows colors
          // PC_NOCOLLAPSE for all others
} PALETTEENTRY;
```

With that in mind, let's look at the palette functions.

Playrix Ex. 1023, page 202

05 0672318350 CH03 05•02•2003 2:34 PM Page 178

Function Prototype:

```
int Set_Palette_Entry(
        int color_index, // color index to change
        LPPALETTEENTRY color); // the color
```

Purpose:

`Set_Palette_Entry()` is used to change a single color in the color palette. You simply send the color index `0..255` along with a pointer to `PALETTEENTRY` holding the color, and the update will occur on the next frame. In addition, this function updates the shadow palette. Note that this function is slow, so if you need to update the entire palette, use `Set_Palette()`. Returns `TRUE` if successful, `FALSE` otherwise.

Example:

```
// set color 0 to black
PALETTEENTRY black = {0,0,0,PC_NOCOLLAPSE};
Set_Palette_Entry(0,&black);
```

Function Prototype:

```
int Get_Palette_Entry(
       int color_index, // color index to retrieve
       LPPALETTEENTRY color); // storage for color
```

Purpose:

`Get_Palette_Entry()` retrieves a palette entry from the current palette. However, the function is very fast because it retrieves the data from the RAM-based shadow palette. Hence, you can call this as much as you like, and it doesn't disturb the hardware at all. However, if you make changes to the system palette with the use of `Set_Palette_Entry()` or `Set_Palette()`, the shadow palette will not be updated and the data retrieved might not be valid. Returns `TRUE` if successful, `FALSE` otherwise.

Example:

```
// let's get palette entry 100
PALETTEENTRY color;
Get_Palette_Entry(100,&color);
```

Function Prototype:

```
int Save_Palette_To_File(
        char *filename, // filename to save at
        LPPALETTEENTRY palette); // palette to save
```

Playrix Ex. 1023, page 203

05 0672318350 CH03 05•02•2003 2:34 PM Page 179

Purpose:

`Save_Palette_To_File()` saves the sent palette data to an ASCII file on disk for later retrieval or processing. This function is very handy if you generate a palette on the fly and want to store it on disk. However, the function assumes that the pointer in the palette points to a 256 entry palette—so watch out! Returns `TRUE` if successful, `FALSE` otherwise.

Example:

```
PALETTEENTRY my_palette[256]; // assume this is built

// save the palette we made
// note file name can be anything, but I like *.pal
Save_Palette_To_File("/palettes/custom1.pal",&my_palette);
```


Function Prototype:

```
int Load_Palette_From_File(
     char *filename, // file to load from
     LPPALETTEENTRY palette); // storage for palette
```

Purpose:

`Load_Palette_From_File()` is used to load a previously saved 256-color palette from disk via `Save_Palette_To_File()`. You simply send the file name along with storage for all 256 entries and the palette is loaded off disk into the data structure. However, the function does NOT load the entries into the hardware palette; you must do this yourself with `Set_Palette()`. Returns `TRUE` if successful, `FALSE` otherwise.

Example:

```
// load the previously saved palette
Load_Palette_From_File("/palettes/custom1.pal",&my_palette);
```

> **NOTE**
>
> When you create a 256-color display mode via a call to `DDRAW_INIT()`, the function will load a standard palette that has almost optimal color space coverage. The name of the palette file is `PALDATA2.DAT`. It's nothing more than an ASCII file with 256 line entries, each with a red, green, and blue value in the range of `0..255`.

Function Prototype:

```
int Set_Palette(LPPALETTEENTRY set_palette); // palette to load into hardware
```

Purpose:

`Set_Palette()` loads the sent palette data into the hardware and updates the shadow palette, also. Returns `TRUE` if successful, `FALSE` otherwise.

Playrix Ex. 1023, page 204

05 0672318350 CH03 05•02•2003 2:34 PM Page 180

Example:

```
// lets load the palette into the hardware
Set_Palette(&my_palette);
```

Function Prototype:

```
int Save_Palette(LPPALETTEENTRY sav_palette); // storage for palette
```

Purpose:

`Save_Palette()` scans the hardware palette out into `sav_palette` so that you can save it to disk or manipulate it. `sav_palette` must have enough storage for all 256 entries.

Example:

```
// retrieve the current DirectDraw hardware palette
PALETTEENTRY hardware_palette[256];
Save_Palette(&hardware_palette);
```

Function Prototype:

```
int Rotate_Colors(int start_index, // starting index 0..255
         int end_index); // ending index 0..255
```

Purpose:

`Rotate_Colors()` rotates a bank of colors in a cyclic manner. It manipulates the color palette hardware directly. Returns `TRUE` if successful, `FALSE` otherwise.

Example:

```
// rotate the entire palette
Rotate_Colors(0,255);
```

Function Prototype:

```
int Blink_Colors(int command, // blinker engine command
         BLINKER_PTR new_light, // blinker data
         int id); // id of blinker
```

Purpose:

`Blink_Colors()` is used to create asynchronous palette animation. The function is too long to explain here, but please refer to the previous chapter for a more in-depth description.

Example:

N/A

Playrix Ex. 1023, page 205

05 0672318350 CH03 05•02•2003 2:34 PM Page 181

Utility Functions

Function Prototype:

```
DWORD Get_Clock(void);
```

Purpose:

`Get_Clock()` returns the current clock time in milliseconds since Windows was started.

Example:

```
// get the current tick count
DWORD start_time = Get_Clock();
```


Function Prototype:

```
DWORD Start_Clock(void);
```

Purpose:

`Start_Clock()` basically makes a call to `Get_Clock()` and stores the time in a global variable for you. Then you can call `Wait_Clock()`, which will wait for some number of milliseconds since your call to `Start_Clock()`. Returns the starting clock value at the time of the call.

Example:

```
// start the clock and set the global
Start_Clock();
```

Function Prototype:

```
DWORD Wait_Clock(DWORD count); // number of milliseconds to wait
```

Purpose:

`Wait_Clock()` simply waits the sent number of milliseconds since the call was made to `Start_Clock()`. Returns the current clock count at the time of call. However, the function will NOT return until the time difference has elapsed.

Example:

```
// wait 30 milliseconds
Start_Clock();

// code…

Wait_Clock(30);
```

Playrix Ex. 1023, page 206

05 0672318350 CH03 05•02•2003 2:34 PM Page 182

NOTE

Later in the book, we will probably upgrade to using the high performance timers that Windows supports for better resolution.

Function Prototype:

```
int Collision_Test(int x1, int y1, // upper lhs of obj1
          int w1, int h1, // width, height of obj1
          int x2, int y2, // upper lhs of obj2
          int w2, int h2);// width, height of obj2
```

Purpose:

`Collision_Test()` basically performs an overlapping rectangle test on the two sent rectangles. The rectangles can represent whatever you like. You must send the upper-left hand corner coordinates of each rectangle, along with its width and height. Returns `TRUE` if there is an overlap, `FALSE` otherwise.

Example:

```
// do these two BITMAP_IMAGE's overlap?
if (Collision_Test(ship1->x,ship1->y,ship1->width,ship1->height,
          ship2->x,ship2->y,ship2->width,ship2->height))
  { // hit

  } // end if
```

Function Prototype:

```
int Color_Scan(int x1, int y1, // upper left of rect
        int x2, int y2, // lower right of rect
        UCHAR scan_start, // starting scan color
        UCHAR scan_end, // ending scan color
        UCHAR *scan_buffer, // memory to scan
        int scan_lpitch); // linear memory pitch
```

Purpose:

`Color_Scan()` is another collision detection algorithm that scans a rectangle for a single 8-bit value or sequence of values in some continuous range. You can use it to determine whether a color index is present within some area. Of course, it only works with 8-bit images, but the source is easily extensible to 16-bit or higher modes. Returns `TRUE` if the color(s) is found.

Playrix Ex. 1023, page 207

05 0672318350 CH03 05•02•2003 2:34 PM Page 183

Example:

```
// scan for colors in range from 122-124 inclusive
Color_Scan(10,10, 50, 50, 122,124, back_buffer, back_lpitch);
```

The BOB (Blitter Object) Engine

Although with a bit of programming you can get the BITMAP_IMAGE type to do what you want, it is lacking in a serious way—it doesn't use DirectDraw surfaces, and hence there is no support for acceleration. Therefore, I have created a type called a *BOB*, or *blitter object,* that is very similar to a sprite. A *sprite* (for those of you that have been in a game programming cave) is nothing more than an object you can move around the screen that usually doesn't disturb the background. In our case, this isn't true; therefore, I called my animation object a BOB rather than a sprite, so there!

We will use the BOB engine very little in this book, but I still wanted you to see it, because it's a good example of using DirectDraw surfaces and full 2D acceleration. So let's talk briefly about what a BOB is. First here's the data structure for a BOB:

```
// the blitter object structure BOB
typedef struct BOB_TYP
    {
    int state;     // the state of the object (general)
    int anim_state;   // an animation state variable, up to you
    int attr;      // attributes pertaining to the object (general)
    float x,y;      // position bitmap will be displayed at
    float xv,yv;     // velocity of object
    int width, height; // the width and height of the bob
    int width_fill;   // internal, used to force 8*x wide surfaces
    int bpp;      // bits per pixel needed for 8/16 bit support
    int counter_1;   // general counters
    int counter_2;
    int max_count_1;  // general threshold values;
    int max_count_2;
    int varsI[16];   // stack of 16 integers
    float varsF[16];  // stack of 16 floats
    int curr_frame;   // current animation frame
    int num_frames;   // total number of animation frames
    int curr_animation; // index of current animation
    int anim_counter;  // used to time animation transitions
    int anim_index;   // animation element index
    int anim_count_max; // number of cycles before animation
    int *animations[MAX_BOB_ANIMATIONS]; // animation sequences
    // the bitmap images DD surfaces
    LPDIRECTDRAWSURFACE7 images[MAX_BOB_FRAMES];
    } BOB, *BOB_PTR;
```

Playrix Ex. 1023, page 208

05 0672318350 CH03 05•02•2003 2:34 PM Page 184

A BOB is basically a graphical object represented by one or more DirectDraw surfaces (up to 64 as currently `#defined`). You can move a BOB, draw a BOB, animate a BOB, and set it in motion. BOBs consider the current DirectDraw clipper, so they are clipped as well as accelerated—which is a good thing! Figure 3.10 shows a BOB and it relationship to its animation frames.

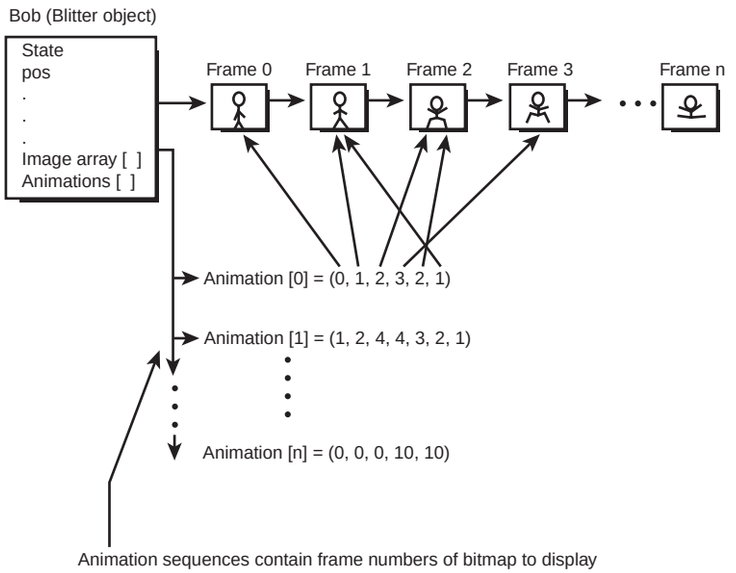


FIGURE 3.10 The BOB animation system.

Also, the BOB engine supports 8- or 16-bit images and animation sequences, so you can load in a set of frames and an animation sequence, and the sequence will play feeding from the frames. This is a very cool feature. Additionally, BOBs work with both 8- and 16-bit color, but most of the functions have internal logic that know what bit depth the system is in, so you don't have to worry it. The only functions that you need to explicitly call a different version for 16-bit mode are `Load_Frame_BOB*()` (when you load) and `Draw_BOB*()` (when you draw a BOB), but we will get to that, along with reviewing examples of both 8- and 16-bit demos, at the end of the chapter.

Finally, all the BOB functions return `TRUE` if successful and `FALSE` otherwise. So let's take a look at them all....

Function Prototype:

```
int Create_BOB(BOB_PTR bob, // ptr to bob to create
  int x, int y,      // initial position of bob
  int width, int height,  // size of bob
```

Playrix Ex. 1023, page 209

05 0672318350 CH03 05•02•2003 2:34 PM Page 185


```
int num_frames, // total number of frames for bob
int attr,    // attributes of bob
int mem_flags=0, // surface memory flags, 0 is VRAM
USHORT color_key_value=0, // color key value is interpreted as
            // 8-bit index, or 16-bit RGB based
            // on the bpp setting below
int bpp=8); // bits per pixel, default is 8
```

Purpose:

`Create_BOB()` creates a single BOB object and sets it all up. The function assigns all the internal variables, in addition to creating a separate DirectDraw surface for each frame. Most of the parameters are self-explanatory—the only value that needs a little explanation is the attribute variable `attr`. Take a look at Table 3.1 to see a better description of each of the attributes you can logically `OR` together and send in this field.

TABLE 3.1 Valid BOB Attributes

Value	Meaning
`BOB_ATTR_SINGLE_FRAME`	Create a BOB with a single frame.
`BOB_ATTR_MULTI_FRAME`	Create a BOB with multiple frames, but the animation of the BOB will be a linear sequence through the frames `0..`*n*.
`BOB_ATTR_MULTI_ANIM`	Create a multiple frame BOB that supports animation sequences.
`BOB_ATTR_ANIM_ONE_SHOT`	If this is set then when an animation sequence plays, the animation sequence will only play once and then stop. At this point, the internal variable `anim_state` will be set. To play the animation again, reset this variable.
`BOB_ATTR_BOUNCE`	This flag tells the BOB to bounce off the screen boundaries like a ball. This only works if you use `Move_BOB()`.
`BOB_ATTR_WRAPAROUND`	This flag tells the BOB to wrap around to the other side of the screen as it moves. This only works if you use `Move_BOB()`.

Example:

Here are some examples creating BOBs. First, a single frame 8-bit BOB at (50,100) with a size of 96×64:

```
BOB car; // a car bob

// create the bob
if (!Create_BOB(&car, 50,100, // address of bob and initial position
        96,64,    // size of bob
        1,      // number of frames
        BOB_ATTR_SINGLE_FRAME, // attribute
        0, // memory flags
```

Playrix Ex. 1023, page 210

05 0672318350 CH03 05•02•2003 2:34 PM Page 186


```
        0, // color key, for 8-bit this is the index
        8)) // bits per pixel, 8 for 256 color modes
  { /* error */ }
```

And here's a multiple frame 16-bit BOB with 8 frames and a size of 32×32:

```
BOB ship; // a space ship bob

// create the bob
if (!Create_BOB(&ship, 0,0, // address of bob and initial position
        32,32, // size of bob
        8,   // number of frames
        BOB_ATTR_MULTI_FRAME, // attribute
        0, // memory flags
        0, // color key, for 16-bit mode this is the RGB word
        16)) // bits per pixel, 16 for 16 bit modes
  { /* error */ }
```

Finally, a multiple frame 8-bit BOB that supports animation sequences:

```
BOB greeny; // a little green man bob

// create the bob
if (!Create_BOB(&greeny, 0,0,
  32,32,32,BOB_ATTR_MULTI_ANIM,0,0,8))
  { /* error */ }
```

TIP

Note that the last three parameters to `Create_BOB()` all have default values. Therefore, you don't need to override them if you don't want to. If 0,0,8 is okay with you, you don't have to type them.

Function Prototype:

```
int Destroy_BOB(BOB_PTR bob); // ptr to bob to destroy
```

Purpose:

`Destroy_BOB()` destroys a previously created BOB. It doesn't matter if it's 8- or 16-bit.

Example:

```
// destroy the BOB above, you would do this
Destroy_BOB(&greeny);
```

Playrix Ex. 1023, page 211

05 0672318350 CH03 05•02•2003 2:34 PM Page 187

Function Prototype:

```
int Draw_BOB(BOB_PTR bob, // ptr of bob to draw
 LPDIRECTDRAWSURFACE7 dest); // dest surface to draw on

// and the 16-bit version
int Draw_BOB16(BOB_PTR bob, // ptr of bob to draw
 LPDIRECTDRAWSURFACE7 dest); // dest surface to draw on
```

Purpose:

`Draw_BOB*()` is a very powerful function. It draws the sent BOB on the DirectDraw surface you send it. The BOB is drawn in its current position and current frame (as defined by its animation parameters). Also, make sure to use the correct version of the function depending on whether your BOB is 8- or 16-bit; otherwise, you might get half a BOB!

CAUTION

For this function to work, the destination surface must NOT be locked.

Example:

```
// this is how you would position a multiframe BOB at
// (50,50) and draw the first frame of it on the back
// surface:
BOB ship; // a space ship bob

// create the bob, 8-bit default
if (!Create_BOB(&ship, 0,0,
        32,32,8,BOB_ATTR_MULTI_FRAME,0))

// load the bob images in..we'll get to this in a bit
// set the position and frame of bob
ship.x = 50;
ship.y = 50;
ship.curr_frame = 0; // this contains the frame to draw

// draw bob
Draw_BOB(&ship, lpddsback);
```

Function Prototype:

```
int Draw_Scaled_BOB(BOB_PTR bob, // ptr of bob to draw
 int swidth, int sheight, // new width and height of bob
 LPDIRECTDRAWSURFACE7 dest); // dest surface to draw on
```

Playrix Ex. 1023, page 212

05 0672318350 CH03 05•02•2003 2:34 PM Page 188


```
// and the 16-bit version
int Draw_Scaled_BOB16(BOB_PTR bob, // ptr of bob to draw
 int swidth, int sheight, // new width and height of bob
 LPDIRECTDRAWSURFACE7 dest); // dest surface to draw on
```

Purpose:

`Draw_Scaled_BOB*()` works exactly like `Draw_BOB()`, except you can send any width and height to draw the BOB and the BOB will be scaled appropriately. This is very cool, and if you have acceleration, it's a great way to scale a BOB to make it look 3D! In fact, we will use a technique like this when we create billboards for our 3D applications.

> **NOTE**
>
> *Billboards* are simply 2D images that are drawn in 3D, but are always perpendicular to the view direction. For example, in *Doom*, all the creatures are billboard sprites, while the world is polygon-based.

Example:

```
// an example of drawing the ship 128x128 even though
// it was created as only 32x32 pixels
// 8-bit version
Draw_Scaled_BOB(&ship, 128,128,lpddsback);
```

Function Prototype:

```
int Load_Frame_BOB(
 BOB_PTR bob, // ptr of bob to load frame into
 BITMAP_FILE_PTR bitmap,// ptr of bitmap file to scan data
 int frame,  // frame number to place image into 0,1,2...
 int cx,int cy, // cell pos or abs pos to scan from
 int mode);   // scan mode, same as Load_Frame_Bitmap()

// and the 16-bit version
int Load_Frame_BOB16(
 BOB_PTR bob, // ptr of bob to load frame into
 BITMAP_FILE_PTR bitmap,// ptr of bitmap file to scan data
 int frame,  // frame number to place image into 0,1,2...
 int cx,int cy, // cell pos or abs pos to scan from
 int mode);   // scan mode, same as Load_Frame_Bitmap()
```

Purpose:

The `Load_Frame_BOB*()` function works identically to the `Load_Frame_Bitmap()` function, so refer to it for details. The only addition is the control of the frame to load via frame. If

Playrix Ex. 1023, page 213

05 0672318350 CH03 05•02•2003 2:34 PM Page 189

you create a BOB that has four frames, you will load the frames one by one. Additionally, you must call the correct version of the function (8- or 16-bit) based on the bit depth of the data and screen.

Example:

```
// here's an example of loading 4 frames into a 16-bit BOB from a
// bitmap file in cell mode

BOB ship; // the bob
// loads frames 0,1,2,3 from cell position (0,0), (1,0),
// (2,0), (3,0)
// from bitmap16bit bitmap file, assume it has been loaded

for (int index=0; index<4; index++)
   Load_Frame_BOB16(&ship,&bitmap16bit,
           index, index,0,
           BITMAP_EXTRACT_MODE_CELL );
```


NOTE

The next set of BOB manipulation functions work on the data structure elements of the BOB, so they work identically on both 8- and 16-bit BOBs.

Function Prototype:

```
int Load_Animation_BOB(
  BOB_PTR bob,  // bob to load animation into
  int anim_index, // which animation to load 0..15
  int num_frames, // number of frames of animation
  int *sequence); // ptr to array holding sequence
```

Purpose:

`Load_Animation()` takes a little explaining. The function is used to load one of 16 arrays internal to the BOB that contain animation sequences. Each sequence contains an array of indices or frame numbers to display in sequence. This is shown at the bottom of Figure 3.10.

Example:

You might have a BOB that has eight frames `0,1,…7`, but you might have four animations defined as follows:

```
int anim_walk[] = {0,1,2,1,0};
int anim_fire[] = {5,6,0};
int anim_die[]  = {3,4};
int anim_sleep[] = {0,0,7,0,0};
```

Playrix Ex. 1023, page 214

05 0672318350 CH03 05•02•2003 2:34 PM Page 190

Then to load the animations into the BOB, you would do this:

```
// create a mutli animation bob
// create the bob
if (!Create_BOB(&alien, 0,0, 32,32,8,BOB_ATTR_MULTI_ANIM,0))
  { /* error */ }

// load the bob frames in...
// load walk into animation 0
Load_Animation_BOB(&alien, 0,5,anim_walk);

// load fire into animation 1
Load_Animation_BOB(&alien, 1,3,anim_fire);

// load die into animation 2
Load_Animation_BOB(&alien, 2,2,anim_die);

// load sleep into animation 3
Load_Animation_BOB(&alien, 3,5,anim_sleep);
```

After loading the animations, you can set the active animation and play them with functions you'll see in a minute.

Function Prototype:

```
int Set_Pos_BOB(BOB_PTR bob, // ptr to bob to set position
       int x, int y); // new position of bob
```

Purpose:

`Set_Pos_BOB()` is a simple way to set the position of the BOB. This does nothing more than assign the internal (x,y) variables, but it's nice to have a function.

Example:

```
// set the position of the alien BOB above
Set_Pos_BOB(&alien, player_x, player_y);
```

Function Prototype:

```
int Set_Vel_BOB(BOB_PTR bob, // ptr to bob to set velocity
     int xv, int yv); // new x,y velocity
```

Purpose:

Each BOB has an internal velocity contained in `(xv,yv)`. `Set_Vel_BOB()` simply assigns these values the new values sent in the function. The velocity values in the BOB won't do

Playrix Ex. 1023, page 215

05 0672318350 CH03 05•02•2003 2:34 PM Page 191

anything unless you use the function Move_BOB() to move your BOBs. However, even if you don't, you can use xv,yv to track the velocity of the BOB yourself.

Example:

```
// make the BOB move in a straight horizontal line
Set_Vel_BOB(&alien, 10,0);
```

Function Prototype:

```
int Set_Anim_Speed_BOB(BOB_PTR bob, // ptr to bob
         int speed); // speed of animation
```


Purpose:

Set_Anim_Speed() sets the internal animation rate for a BOB anim_count_max. The higher this number is, the slower the animation; the lower the number (0 is the lowest), the faster the animation. However, this function only matters if you use the internal BOB animation function Animate_BOB(). And of course, you must have created a BOB that has multiple frames.

Example:

```
// set the rate to change frames every 30 frames
Set_Anim_Speed_BOB(&alien, 30);
```

Function Prototype:

```
int Set_Animation_BOB(
     BOB_PTR bob, // ptr of bob to set animation
     int anim_index); // index of animation to set
```

Purpose:

Set_Animation_BOB() sets the current animation that will be played by the BOB. In the earlier example of Load_Animation_BOB(), we created four animations.

Example:

```
// make animation sequence number 2 active
Set_Animation_BOB(&alien, 2);
```

> **NOTE**
>
> This also resets the BOB animation to the first frame in the sequence.

Function Prototype:

```
int Animate_BOB(BOB_PTR bob); // ptr to bob to animate
```

Playrix Ex. 1023, page 216

05 0672318350 CH03 05•02•2003 2:34 PM Page 192

Purpose:

`Animate_BOB()` animates a BOB for you. Normally, you would call this function once a frame to update the animation of the BOB.

Example:

```
// erase everything...
// move everything...
// animate everything
Animate_BOB(&alien);
```

Function Prototype:

```
int Move_BOB(BOB_PTR bob); // ptr of bob to move
```

Purpose:

`Move_BOB()` moves the BOB a delta of `xv,yv` (set with the `Set_Vel_BOB()` function), then, depending on the attributes, will either bounce the BOB off the walls, wrap it around, or do nothing. Similarly to the `Animate_BOB()` function, you would place this call once in the main loop right after (or before) `Animate_BOB()`.

Example:

```
// animate bob
Animate_BOB(&alien);

// move it
Move_BOB(&alien);
```

Function Prototype:

```
int Hide_BOB(BOB_PTR bob); // ptr to bob to hide
```

Purpose:

`Hide_BOB()` simply sets the BOB's internal visibility flag to `0`, after which `Draw_BOB()` won't display the BOB anymore.

Example:

```
// hide the bob
Hide_BOB(&alien);
```

Function Prototype:

```
int Show_BOB(BOB_PTR bob); // ptr to bob to show
```

Playrix Ex. 1023, page 217

05 0672318350 CH03 05•02•2003 2:34 PM Page 193

Purpose:

`Show_BOB()` sets the visibility flag to 1 on a BOB, so it will be drawn (it undoes a `Hide_BOB()` call). Here's an example of hiding and showing a BOB because you are displaying a GDI object or something, and don't want the BOB to occlude it:

Example:

```
// Hide_BOB(&alien);
// make calls to Draw_BOB and GDI etc.
Show_BOB(&alien);
```


Function Prototype:

```
int Collision_BOBS(BOB_PTR bob1, // ptr to first bob
          BOB_PTR bob2); // ptr to second bob
```

Purpose:

`Collision_BOBS()` detects whether the bounding rectangles of two BOBs are overlapping. This can be used for collision detection in a game to see whether a player BOB hits a missile BOB or whatever.

Example:

```
// check if a missile BOB hit a player BOB:
if (Collision_BOBS(&missile, &player))
  { /* make explosion sound */ }
```

That's about it for the graphics module of the `T3DLIB` game library (`T3DLIB1.CPP¦H`). Now let's move on to the sound module.

The T3DLIB2 DirectX Input System

Writing a simple set of wrapper functions around DirectInput is almost a no-brainer. Well, it takes some brains, but for the most part it's fairly easy. All we need to do is create an API with a very simple interface and few parameters. The interface should support the following minimal functionality:

- Initialize the DirectInput system
- Set up and acquire the keyboard, mouse, and joystick—or any subset
- Read data from any of the input devices
- Shut down, unacquire, and release everything

I have created such an API, and it's available in `T3DLIB2.CPP¦H` on the CD. The API does everything you need to initialize DirectInput and read any device. Before reviewing the

Playrix Ex. 1023, page 218

05 0672318350 CH03 05•02•2003 2:34 PM Page 194

functions, take a look at Figure 3.11—it depicts the relationship between each device and the data flow.

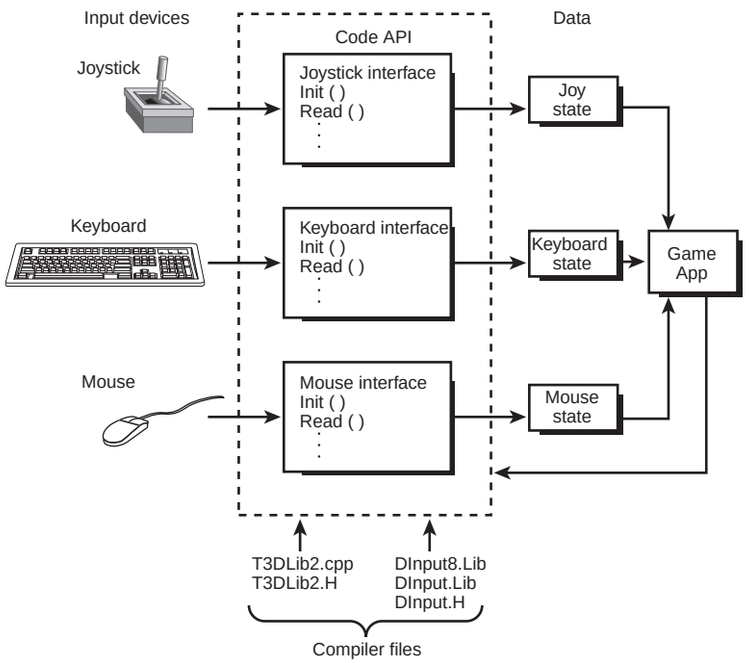


FIGURE 3.11 The DirectInput software system.

Here are the globals for the library:

```
LPDIRECTINPUT8    lpdi;    // dinput object
LPDIRECTINPUTDEVICE8 lpdikey;  // dinput keyboard
LPDIRECTINPUTDEVICE8 lpdimouse; // dinput mouse
LPDIRECTINPUTDEVICE8 lpdijoy;  // dinput joystick
GUID  joystickGUID; // guid for main joystick
char  joyname[80]; // name of joystick

// all input is stored in these records
UCHAR keyboard_state[256]; // contains keyboard state table
DIMOUSESTATE mouse_state; // contains state of mouse
DIJOYSTATE joy_state;   // contains state of joystick
int joystick_found;    // tracks if stick is plugged in
```

Input from the keyboard is placed in `keyboard_state[]`, the mouse data is stored in `mouse_state`, and the joystick data is stored in `joy_state` by the input system. The

Playrix Ex. 1023, page 219

05 0672318350 CH03 05•02•2003 2:34 PM Page 195

structure of each of these records is the standard DirectInput device state structures (except for the keyboard; it's just a Boolean array of BYTES) as shown in the following:

```
// for the mouse data
typedef struct DIMOUSESTATE {
  LONG lX; // x-axis
  LONG lY; // y-axis
  LONG lZ; // z-axis
  BYTE rgbButtons[4]; // state of the buttons
} DIMOUSESTATE, *LPDIMOUSESTATE;

// for the joystick data
typedef struct DIJOYSTATE {
  LONG  lX; // x-axis
  LONG  lY; // y-axis
  LONG  lZ; // z-axis
  LONG  lRx; // rotation about x-axis
  LONG  lRy; // rotation about y-axis
  LONG  lRz; // rotation about z-axis
  LONG  rglSlider[2]; // u,v slider positions
  DWORD  rgdwPOV[4];  // point of view hat state
  BYTE  rgbButtons[32]; // state of buttons 0..31
} DIJOYSTATE, *LPDIJOYSTATE;
```

In general, the mouse and joystick are roughly equivalent as far as the x,y position goes; that is, you access them via the fields lX, lY, and the buttons are BOOLEANs in rgbButtons[]. Let's get to the functions. The variable joystick_found is a BOOLEAN that is set when you request joystick access. If a joystick is found, it is TRUE, otherwise, it's FALSE. With it, you can conditional block out code that uses the joystick. So without further ado, here is the new API:

Function Prototype:

```
int DInput_Init(void);
```

Purpose:

DInput_Init() initializes the DirectInput input system. It creates the main COM object and returns TRUE if successful and FALSE otherwise. And of course, the global lpdi will be valid. The function does not create any devices, though. Here's an example of initializing the input system:

Example:

```
if (!DInput_Init())
  { /* error */ }
```

Playrix Ex. 1023, page 220

05 0672318350 CH03 05•02•2003 2:34 PM Page 196

Function Prototype:

```
void DInput_Shutdown(void);
```

Purpose:

`DInput_Shutdown()` releases all the COM objects and any resources allocated during the call to `DInput_Init()`. Normally, you would call `DInput_Shutdown()` at the very end of your application after you have released all the input devices themselves—which we'll get to shortly. Anyway, here's an example of shutting down the input system:

Example:

```
DInput_Shutdown();
```

Function Prototype:

```
DInput_Init_Keyboard(void);
```

Purpose:

`DInput_Init_Keyboard()` initializes and acquires the keyboard. This should always work and return `TRUE`, unless another DirectX application has taken over in a really uncooperative way. Here's an example:

Example:

```
if (!DInput_Init_Keyboard())
  { /* error */ }
```

Function Prototype:

```
int DInput_Init_Mouse(void);
```

Purpose:

`DInput_Init_Mouse()` initializes and acquires the mouse. The function takes no parameters and returns `TRUE` if successful and `FALSE` otherwise. But it should always work, unless a mouse isn't plugged in or there's another DirectX application that has totally taken over! If everything goes well, `lpdimouse` becomes the valid interface pointer. Here's an example:

Example:

```
if (!DInput_Init_Mouse()) { /* error */ }
```

Function Prototype:

```
int DInput_Init_Joystick(int min_x=-256, // min x range
        int max_x=256, // max x range
        int min_y=-256, // min y range
        int max_y=256, // max y range
        int dead_zone=10); // dead zone in percent
```

Playrix Ex. 1023, page 221

05 0672318350 CH03 05•02•2003 2:34 PM Page 197

Purpose:

`DInput_Init_Joystick()` initializes the joystick device for use. The function takes five parameters that define the x-y range of motion of the data sent back from the joystick and the dead zone with a percentage. If you want to use the defaults of -256 to 256 and 10% dead zone for each axis, you don't need to send parameters because they have default values (C++ thing). If the call returns back a `TRUE`, a joystick was found and it has been set up, initialized, and acquired. After the call, the interface pointer `lpdijoy` will be valid if you need it for anything. In addition, the string `joyname[]` will contain the "friendly" name of the joystick device, such as "Microsoft Sidewinder Pro," and so on.

Here's an example of initializing the joystick and setting its x-y ranges to -1024 to 1024 with a 5% dead zone:

Example:

```
if (!DInput_Init_Joystick(-1024, 1024, -1024, 1024, 5))
  { /* error */ }
```

Function Prototype(s):

```
void DInput_Release_Joystick(void);
void DInput_Release_Mouse(void);
void DInput_Release_Keyboard(void);
```

Purpose:

`DInput_Release_Joystick()`, `DInput_Release_Mouse()`, and `DInput_Release_Keyboard()` release each of the respective input devices when you are done with them. The functions can be called even if you haven't initialized the respective device, so you can just call them all at the end of your application if you want. Here's a complete example of starting up the DirectInput system, initializing all the devices and then releasing them and shutting down:

Example:

```
// initialize the DirectInput system
DInput_Init();

// initialize all input devices and acquire them
DInput_Init_Joystick();
DInput_Init_Mouse();
DInput_Init_Keyboard();

// input loop ....do work here
// now done...
```

Playrix Ex. 1023, page 222

05 0672318350 CH03 05•02•2003 2:34 PM Page 198


```
// first release all devices, order is unimportant
DInput_Release_Joystick();
DInput_Release_Mouse();
DInput_Release_Keyboard();

// shutdown DirectInput
DInput_Shutdown();
```

Function Prototype:

```
int DInput_Read_Keyboard(void);
```

Purpose:

`DInput_Read_Keyboard()` scans the keyboard state and places the data in `keyboard_state[]`, which is an array of 256 `BYTE`s. This is the standard DirectInput keyboard state array, so you must use the DirectInput key constants `DIK_*` if you want to make sense of it. If a key is pressed, the array value will be `0x80`. Here's an example of testing whether the right and left keys are down using the manifest constants in DirectInput—which you can look up in the SDK or the abridged Table 3.2 following.

Example:

```
// read the keyboard
if (!DInput_Read_Keyboard())
  { /* error */ }

// now test the state data
if (keyboard_state[DIK_RIGHT])
  { /* move ship right */ }
else
if (keyboard_state[DIK_LEFT])
  { /* move ship left */ }
```

TABLE 3.2 The DirectInput Keyboard State Constants

Symbol	Meaning
`DIK_ESCAPE`	The Esc key
`DIK_0-9`	Main keyboard 0–9
`DIK_MINUS`	Minus key
`DIK_EQUALS`	Equals key
`DIK_BACK`	Backspace key
`DIK_TAB`	Tab key
`DIK_A-Z`	Letters A–Z
`DIK_LBRACKET`	Left bracket
`DIK_RBRACKET`	Right bracket

Playrix Ex. 1023, page 223

05 0672318350 CH03 05•02•2003 2:34 PM Page 199

TABLE 3.2 Continued

Symbol	Meaning
DIK_RETURN	Return/Enter on main keyboard
DIK_LCONTROL	Left control
DIK_LSHIFT	Left shift
DIK_RSHIFT	Right shift
DIK_LMENU	Left Alt
DIK_SPACE	Spacebar
DIK_F1-15	Function keys 1–15
DIK_NUMPAD0-9	Numeric keypad keys 0–9
DIK_ADD	+ on numeric keypad
DIK_NUMPADENTER	Enter on numeric keypad
DIK_RCONTROL	Right control
DIK_RMENU	Right Alt
DIK_HOME	Home on arrow keypad
DIK_UP	UpArrow on arrow keypad
DIK_PRIOR	PgUp on arrow keypad
DIK_LEFT	LeftArrow on arrow keypad
DIK_RIGHT	RightArrow on arrow keypad
DIK_END	End on arrow keypad
DIK_DOWN	DownArrow on arrow keypad
DIK_NEXT	PgDn on arrow keypad
DIK_INSERT	Insert on arrow keypad
DIK_DELETE	Delete on arrow keypad

(Note: bolded entries simply mean to follow the sequence. For example, DIK_0-9 means that there are constants DIK_0, DIK_1, DIK_2, and so forth.)

Function Prototype:

```
int DInput_Read_Mouse(void);
```

Purpose:

`DInput_Read_Mouse()` reads the relative mouse state and stores the result in `mouse_state`, which is a `DIMOUSESTATE` structure. The data is in relative delta mode. In most cases, you'll only need to look at `mouse_state.lX`, `mouse_state.lY`, and `rgbButtons[0..2]`, which are `BOOLEAN`s for the three mouse buttons. Here's an example of reading the mouse and using it to move a cursor around and draw:

Example:

```
// read the mouse
if (!DInput_Read_Mouse())
  { /* error */ }
```

Playrix Ex. 1023, page 224

05 0672318350 CH03 05•02•2003 2:34 PM Page 200


```
// move cursor
cx+=mouse_state.lX;
cy+=mouse_state.lY;

// test if left button is down
if (mouse_state.rgbButtons[0])
  Draw_Pixel(cx,cy,col,buffer,pitch);
```

Function Prototype:

```
int DInput_Read_Joystick(void);
```

Purpose:

`DInput_Read_Joystick()` polls the joystick and then reads the data into `joy_state`, which is a `DIJOYSTATE` structure. Of course, if there isn't a joystick plugged in, the function returns `FALSE` and `joy_state` will be invalid, but you get the idea. If successful, `joy_state` contains the state information of the joystick. The data returned will be in the range you previously set for each axis, and the button values are `BOOLEAN`s in `rgbButtons[]`. As an example, here's how you would use the joystick to move a ship right and left and the first button to fire:

Example:

```
// read the joystick data
if (!DInput_Read_Joystick())
  { /* error */ }

// move the ship
ship_x+=joy_state.lX;
ship_y+=joy_state.lY;

// test for trigger
if (joy_state.rgbButtons[0])
  { // fire weapon // }
```

Of course, your joystick might have a lot of buttons and multiple axes. In this case, you can use the other fields of `joy_state` as defined in the `DIJOYSTATE` DirectInput structure.

NOTE

Even though we are using the `IDIRECTINPUTDEVICE8` interface, we don't need to use the `DIJOYSTATE2` structure. This is only for force feedback devices.

Playrix Ex. 1023, page 225

05 0672318350 CH03 05•02•2003 2:34 PM Page 201

At this point, we are ready to take a look at one of the subsystems that most game programmers hate to deal with—sound and music! But under DirectX, along with my wrapper functions, making sound effects and playing music is so easy it's actually fun!

The T3DLIB3 Sound and Music Library

I have taken all the sound and music technology from the first *Tricks* and used it to create the last component to our game engine T3DLIB3. It is composed of two main source files:

- T3DLIB3.CPP—The main C/C++ source
- T3DLIB3.H—The header file

However, you'll also need to include the DirectSound import library DSOUND.LIB to make anything link. However, DirectMusic does *not* have an import library because it's a pure COM object, so there isn't a DMUSIC.LIB. On the other hand, you still need to point your compiler to the DirectSound and DirectMusic .H header files, so it can find them during compilation. Just to remind you, they are:

- DSOUND.H—The standard DirectSound header.
- DMKSCTRL.H—All of these are for DirectMusic.
- DMUSICI.H
- DMUSICC.H
- DMUSICF.H

With all that in mind, let's take a look at the main elements of the T3DLIB3.H header file.

NOTE

With DirectX 8.0+, Microsoft integrated DirectSound and DirectMusic much more tightly and called it DirectAudio. I see no point in changing to DirectAudio, so we will simply use each component separately in this book.

The Header

The header file T3DLIB3.H contains the types, macros, and externals for T3DLIB3.CPP. Here are the #defines you'll find in the header:

```
#define DM_NUM_SEGMENTS 64 // number of midi segments that can be cached in memory

// midi object state defines
#define MIDI_NULL   0  // this midi object is not loaded
#define MIDI_LOADED  1  // this midi object is loaded
```

Playrix Ex. 1023, page 226

05 0672318350 CH03 05•02•2003 2:34 PM Page 202


```
#define MIDI_PLAYING 2  // this midi object is loaded and playing
#define MIDI_STOPPED 3  // this midi object is loaded, but stopped

#define MAX_SOUNDS   256 // max number of sounds in system at once

// digital sound object state defines
#define SOUND_NULL   0 // " "
#define SOUND_LOADED  1
#define SOUND_PLAYING 2
#define SOUND_STOPPED 3
```

Not much for macros—this is just a macro to help convert from 0–100 to the Microsoft decibels scale, and one to convert multibyte characters to wide characters:

```
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100 - volume)))

// Convert from multibyte format to Unicode using the following macro
#define MULTI_TO_WIDE( x,y )
     MultiByteToWideChar( CP_ACP,MB_PRECOMPOSED, y,-1,x,_MAX_PATH)
```

CAUTION

The column width of this book might be too small to fit the whole macro, so the definition might be on two lines. This is a no-no in real life. Macros must be on a single line!

Next up are the types for the sound engine. First, the DirectSound object.

The Types

There are only two types for the sound engine: one to hold a digital sample, and the other to hold a MIDI segment:

```
// this holds a single sound
typedef struct pcm_sound_typ
  {
  LPDIRECTSOUNDBUFFER dsbuffer; // the directsound buffer
                 // containing the sound
  int state;  // state of the sound
  int rate;  // playback rate
  int size;  // size of sound
  int id;   // id number of the sound
  } pcm_sound, *pcm_sound_ptr;
```

Playrix Ex. 1023, page 227

05 0672318350 CH03 05•02•2003 2:34 PM Page 203

And now the DirectMusic segment type:

```
// directmusic MIDI segment
typedef struct DMUSIC_MIDI_TYP
{
IDirectMusicSegment    *dm_segment; // the directmusic segment
IDirectMusicSegmentState  *dm_segstate; // the state of the segment
int            id;      // the id of this segment
int            state;    // state of midi song

} DMUSIC_MIDI, *DMUSIC_MIDI_PTR;
```


Both sounds and MIDI segments will be stored by the engine in the preceding two structures, respectively. Now let's take a look at the globals.

Global Domination

T3DLIB3 contains a number of globals. Let's take a look. First, here are the globals for the DirectSound system:

```
LPDIRECTSOUND  lpds;  // directsound interface pointer
DSBUFFERDESC  dsbd;  // directsound description
DSCAPS    dscaps; // directsound caps
HRESULT    dsresult // general directsound result
DSBCAPS    dsbcaps; // directsound buffer caps

pcm_sound  sound_fx[MAX_SOUNDS]; // array of sound buffers
WAVEFORMATEX  pcmwf; // generic waveformat structure
```

And here are the globals for DirectMusic:

```
// direct music globals
IDirectMusicPerformance  *dm_perf ;  // the directmusic performance manager
IDirectMusicLoader     *dm_loader; // the directmusic loader

// this hold all the directmusic midi objects
DMUSIC_MIDI     dm_midi[DM_NUM_SEGMENTS];
int dm_active_id;  // currently active midi segment
```

NOTE

I have highlighted lines in the arrays that hold sounds and MIDI segments.

Playrix Ex. 1023, page 228

05 0672318350 CH03 05•02•2003 2:34 PM Page 204

You shouldn't have to mess with any of these globals, except to access the interfaces directly if you want. In general, the API will handle everything for you, but the globals are there if you want to tear them up.

There are two parts to the library: DirectSound and DirectMusic. Let's take a look at DirectSound first.

The DirectSound API Wrapper

DirectSound can be complicated or simple, depending on how you use it. If you want a "do it all" API, you're going to end up simply using most of the DirectSound functions themselves. But if you want a simpler API that enables you to initialize DirectSound and load and play sounds of a specific format, that is a lot easier to wrap up in a few functions.

So, what I've done is take much of our work in setting up DirectSound and formalize it into a set of functions for you. In addition, I've created an abstraction around the sound system, so you refer to a sound with an ID (same for the DirectMusic part) that is given to you during the loading process. Thus, you can use this ID to play the sound, check its status, or to terminate it. This way, there aren't any ugly interface pointers that you have to mess with. The new API supports the following functionality:

- Initializing and shutting down DirectSound with single calls
- Loading .WAV files with 11KHz 8-bit mono format
- Playing a loaded sound file
- Stopping a sound
- Testing the play status of a sound
- Changing the volume, playback rate, or stereo panning of a sound
- Deleting sounds from memory

Let's take a look at each function one by one.

> **NOTE**
>
> Unless otherwise stated, all functions return `TRUE` (`1`) if successful and `FALSE` (`0`) if not.

Function Prototype:

```
int DSound_Init(void);
```

Purpose:

`DSound_Init()` is used to initialize the entire DirectSound system. It creates the DirectSound COM object, sets the priority level, and so forth. Just call the function at the beginning of your application if you want to use sound. Here's an example:

Playrix Ex. 1023, page 229

05 0672318350 CH03 05•02•2003 2:34 PM Page 205

Example:

```
if (!DSound_Init(void))
  { /* error */ }
```

Function Prototype:

```
int DSound_Shutdown(void);
```

Purpose:

`DSound_Shutdown()` is used to shut down and release all the COM interfaces created during `DSound_Init()`. However, `DSound_Shutdown()` will not release all the memory allocated to all the sound. You must do this yourself with another function. Anyway, here's how you would shut down DirectSound:

Example:

```
if (!DSound_Shutdown())
  { /* error */ }
```

Function Prototype(s):

```
int DSound_Load_WAV(char *filename);
```

Purpose:

`DSound_Load_WAV()` creates a DirectSound buffer, loads the sound data file into memory, and prepares the sound to be played. The function takes the complete path and filename of the sound file to be loaded (including the extension .WAV) and loads the file off disk. If successful, the function returns a non-negative ID number. You must save this number, because it is used as a handle to reference the sound. If the function can't find the file, or too many sounds are loaded, it will return `-1`. Here's an example of loading a .WAV file named `FIRE.WAV`:

Example:

```
int fire_id = DSound_Load_WAV("FIRE.WAV");

// test for error
if (fire_id==-1)
  { /* error */}
```

Of course, it's up to you on how you want to save the IDs. You might want to use an array or something else.

Finally, you might wonder where the sound data is, and how to mess with it. If you really must, then you can access the data within the `pcm_sound` array `sound_fx[]` using the ID

Playrix Ex. 1023, page 230

05 0672318350 CH03 05•02•2003 2:34 PM Page 206

you get back from either load function as the index. For example, here's how you would access the DirectSound buffer for the sound with ID `sound_id`:

Example:

```
sound_fx[sound_id].dsbuffer
```

Function Prototype:

```
int DSound_Replicate_Sound(int source_id); // id of sound to copy
```

Purpose:

`DSound_Replicate_Sound()` is used to copy a sound without copying the memory used to hold the sound. For example, suppose you have a gunshot sound, and you want to fire three gunshots, each right after the other? The only way to do this right now would be to load three copies of the gunshot sound into three different DirectSound memory buffers, which would be a waste of memory.

Alas, there is a solution—it's possible to create a duplicate or replicant (if you're a *Blade Runner* fan) of the sound buffer, except for the actual sound data. Instead of copying it, we just point a pointer to it, and DirectSound is smart enough to use it as a "source" for multiple sounds using the same data. The bottom line is, if you want to play a gunshot up to eight times, for example, you should load the gunshot once, make seven copies of it, and acquire a total of eight unique IDs. Replicated sounds work exactly the same as normal sounds, but instead of using `DSound_Load_WAV()` to load and create them, you copy them with `DSound_Replicate_Sound()`. Get it? Good! I'm starting to get dizzy! Here's an example of creating eight gunshots:

Example:

```
int gunshot_ids[8]; // this holds all the id's

// load in the master sound
gunshot_ids[0] = Load_WAV("GUNSHOT.WAV");

// now make copies
for (int index=1; index<8; index++)
  gunshot_ids[index] = DSound_Replicate_Sound(gunshot_ids[0]);

// use gunshot_ids[0..7] anyway you which they all go bang!
```

Function Prototype:

```
int DSound_Play_Sound(int id,    // id of sound to play
           int flags=0, // 0 or DSBPLAY_LOOPING
           int volume=0, // unused
           int rate=0,  // unused
           int pan=0);  // unused
```

Playrix Ex. 1023, page 231

05 0672318350 CH03 05•02•2003 2:34 PM Page 207

Purpose:

DSound_Play_Sound() plays a previously loaded sound. You simply send the ID of the sound along with the play flags—0 for a single time, or DSBPLAY_LOOPING to loop it, and the sound will start playing. If the sound is already playing, it will restart at the beginning. Here's an example of loading and playing a sound:

Example:

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id,0);
```

I could have also left out the 0 for flags entirely, because its default parameter is 0:

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id);
```

Either way, the FIRE.WAV sound will play once and then stop. To make it loop, you would send DSBPLAY_LOOPING for the flags parameter.

Function Prototype(s):

```
int DSound_Stop_Sound(int id);
int DSound_Stop_All_Sounds(void);
```

Purpose:

DSound_Stop_Sound() is used to stop a single sound from playing (if it is playing). You simply send the ID of the sound, and that's it. DSound_Stop_All_Sounds() will stop all the sounds currently playing. Here's an example of stopping the fire_id sound:

Example:

```
DSound_Stop_Sound(fire_id);
```

At the end of your program, it's a good idea to stop all the sounds from playing before exiting. You could do this with separate calls to DSound_Stop_Sound() for each sound, or a single call to DSound_Stop_All_Sounds():

```
//...system shutdown code
DSound_Stop_All_Sounds();
```

Function Prototype:

```
int DSound_Delete_Sound(int id); // id of sound to delete
int DSound_Delete_All_Sounds(void);
```

Playrix Ex. 1023, page 232

05 0672318350 CH03 05•02•2003 2:34 PM Page 208

Purpose:

`DSound_Delete_Sound()` deletes a sound from memory and releases the DirectSound buffer associated with it. If the sound is playing, the function will stop it first. `DSound_Delete_All_Sounds()` deletes all previously loaded sounds. Here's an example of deleting the `fire_id` sound:

Example:

```
DSound_Delete_Sound(fire_id);
```

Function Prototype:

```
int DSound_Status_Sound(int id);
```

Purpose:

`DSound_Status_Sound()` tests the status of a loaded sound based on its ID. All you do is pass the function the ID number of the sound, and the function will return one of these values:

- `DSBSTATUS_LOOPING`—The sound is currently playing and is in loop mode.
- `DSBSTATUS_PLAYING`—The sound is currently playing and is in single play mode.

If the value returned from `DSound_Status_Sound()` is neither of the preceding constants, the sound is not playing. Here's a complete example that waits until a sound has completed playing and then deletes it.

Example:

```
// initialize DirectSound
DSound_DSound_Init();

// load a sound
int fire_id = DSound_Load_WAV("FIRE.WAV");

// play the sound in single mode
DSound_Play_Sound(fire_id);

// wait until the sound is done
while(DSound_Sound_Status(fire_id) &
       (DSBSTATUS_LOOPING ¦ DSBSTATUS_PLAYING));

// delete the sound
DSound_Delete_Sound(fire_id);
```

Playrix Ex. 1023, page 233

05 0672318350 CH03 05•02•2003 2:34 PM Page 209


```
// shutdown DirectSound
DSound_DSound_Shutdown();
```

Pretty cool, huh? A lot better than the couple hundred or so lines of code to do it manually with DirectSound!

Function Prototype:

```
int DSound_Set_Sound_Volume(int id,  // id of sound
          int vol); // volume from 0-100
```

Purpose:

`DSound_Set_Sound_Volume()` changes the volume of a sound in real time. Send the ID of the sound along with a value from 0–100 and the sound will change instantly. Here's an example of changing the volume of a sound to 50% of what it was loaded as:

Example:

```
DSound_Set_Sound_Volume(fire_id, 50);
```

You can always change the volume back to 100% like this:

```
DSound_Set_Sound_Volume(fire_id, 100);
```

Function Prototype:

```
int DSound_Set_Sound_Freq(
        int id,  // sound id
        int freq); // new playback rate from 0-100000
```

Purpose:

`DSound_Set_Sound_Freq()` changes the playback frequency of the sound. Because all sounds must be loaded at 11KHz mono, here's how you would double the perceived playback rate:

Example:

```
DSound_Set_Sound_Freq(fire_id, 22050);
```

And to make you sound like Darth Vader, do this:

```
DSound_Set_Sound_Freq(fire_id, 6000);
```

Function Prototype:

```
int DSound_Set_Sound_Pan(
  int id,  // sound id
  int pan); // panning value from -10000 to 10000
```

Playrix Ex. 1023, page 234

05 0672318350 CH03 05•02•2003 2:34 PM Page 210

Purpose:

`DSound_Set_Sound_Pan()` sets the relative intensity of the sound on the right and left speakers. A value of -10,000 is hard left and 10,000 is hard right. If you want equal power, set the pan to `0`. Here's how you would set the pan all the way to the right side:

Example:

```
DSound_Set_Sound_Pan(fire_id, 10000);
```

The DirectMusic API Rapper

Did you get my joke with the title? The DirectMusic API is even simpler than the DirectSound API. I have created functions to initialize DirectMusic, create all the COM objects for you, and enable you to focus on loading and playing MIDI files. Here's the basic functionality list:

- Initialize and shut down DirectMusic with single calls
- Load MIDI files off disk
- Play a MIDI file
- Stop a MIDI that is currently playing
- Test the play status of a MIDI segment
- Automatically connect to DirectSound if DirectSound has previously been initialized
- Delete MIDI segments from memory

Let's take a look at each function one by one.

> **NOTE**
>
> Unless otherwise stated, all functions return `TRUE` (`1`) if successful, and `FALSE` (`0`) if not.

Function Prototype:

```
int DMusic_Init(void);
```

Purpose:

`Dmusic_Init()` initializes DirectMusic and creates all necessary COM objects. You make this call before any other calls to the DirectMusic library. In addition, if you want to use DirectSound, make sure to initialize DirectSound before calling `DMusic_Init()`. Here's an example of using the function:

Playrix Ex. 1023, page 235

05 0672318350 CH03 05•02•2003 2:34 PM Page 211

Example:

```
if (!DMusic_Init())
  { /* error */ }
```

Function Prototype:

```
int DMusic_Shutdown(void);
```

Purpose:

`DMusic_Shutdown()` shuts down the entire DirectMusic engine. It releases all COM objects, in addition to unloading all loaded MIDI segments. Call this function at the end of your application, but before the call to shutdown DirectSound—if you have DirectSound support. Here's an example:

Example:

```
if (!DMusic_Shutdown())
  { /* error */ }

// now shutdown DirectSound…
```

Function Prototype:

```
int DMusic_Load_MIDI(char *filename);
```

Purpose:

`DMusic_Load_MIDI()` loads a MIDI segment into memory and allocates a record in the `midi_ids[]` array. The function returns the ID of the loaded MIDI segment, or `-1` if not successful. The returned ID is used as a reference for all other calls. Here's an example of loading a couple of MIDI files:

Example:

```
// load files
int explode_id = DMusic_Load_MIDI("explosion.mid");
int weapon_id = DMusic_Load_MIDI("laser.mid");

// test files
if (explode_id == -1 ¦¦ weapon_id == -1)
  { /* there was a problem */ }
```

Function Prototype:

```
int DMusic_Delete_MIDI(int id);
```

Playrix Ex. 1023, page 236

05 0672318350 CH03 05•02•2003 2:34 PM Page 212

Purpose:

`DMusic_Delete_MIDI()` deletes a previously loaded MIDI segment from the system. Simply supply the ID to delete. Here's an example of deleting the previously loaded MIDI files in the preceding example:

Example:

```
if (!DMusic_Delete_MIDI(explode_id) ¦¦
  !DMusic_Delete_MIDI(weapon_id) )
{ /* error */ }
```

Function Prototype:

```
int DMusic_Delete_All_MIDI(void);
```

Purpose:

`DMusic_Delete_All_MIDI()` simply deletes all MIDI segments from the system in one call. Here's an example:

Example:

```
// delete both of our segments
if (!DMusic_Delete_All_MIDI())
  { /* error */ }
```

Function Prototype:

```
int DMusic_Play(int id);
```

Purpose:

`DMusic_Play()` plays a MIDI segment from the beginning. Simply supply the ID of the segment you want to play. Here's an example:

Example:

```
// load file
int explode_id = DMusic_Load_MIDI("explosion.mid");

// play it
if (!DMusic_Play(explode_id))
  { /* error */ }
```

Function Prototype:

```
int DMusic_Stop(int id);
```

Playrix Ex. 1023, page 237

05 0672318350 CH03 05•02•2003 2:34 PM Page 213

Purpose:

`DMusic_Stop()` stops a currently playing segment. If the segment is already stopped, the function has no effect. Here's an example:

Example:

```
// stop the laser blast
if (!DMusic_Stop(weapon_id))
  { /* error */ }
```

Function Prototype:

```
int DMusic_Status_MIDI(int id);
```

Purpose:

`DMusic_Status()` tests the status of any MIDI segment based on its ID. The status codes are as follows:

```
#define MIDI_NULL   0  // this midi object is not loaded
#define MIDI_LOADED  1  // this midi object is loaded
#define MIDI_PLAYING 2  // this midi object is loaded and playing
#define MIDI_STOPPED 3  // this midi object is loaded, but stopped
```

Here's an example of changing state based on a MIDI segment completing:

Example:

```
// main game loop
while(1)
   {
   if (DMusic_Status(explode_id) == MIDI_STOPPED)
    game_state = GAME_MUSIC_OVER;

   } // end while
```

That's it for the DirectSound and DirectMusic APIs. As I said, later in the chapter you will see demo examples of both, but for now, let's continue with putting together the final T3D Game Console and bring our virtual computer interface all together in one glorious …Whoooaaa! I'm getting a little carried away here. I'm mean, let's finish up.

The `T3DLIB` Library at a Glance

At this point, we have three main .CPP|H modules that make up the `T3D` library:

- `T3DLIB1.CPP¦H`—DirectDraw plus graphics algorithms
- `T3DLIB2.CPP¦H`—DirectInput
- `T3DLIB3.CPP¦H`—DirectSound & DirectMusic

Playrix Ex. 1023, page 238

05 0672318350 CH03 05•02•2003 2:34 PM Page 214

Keep this in mind when compiling programs. If you want to compile a demo program, call it `DEMOX_Y.CPP`, and then look at its .H includes. If you see it include any of the preceding related .H library modules, you'll obviously need to include the .CPP files, too.

To compile any of the programs, make sure to include both the library source files, as well as all the DirectX .LIB files—and please, for the sake of God, set your compiler to Win32 .EXE as the target!

Building the Final `T3D` Game Console

At this point, we have more than enough to implement our virtual graphics interface, including support for sound and input. Moreover, the virtual interface model can almost be implemented one-to-one with the data structures, globals, and functions you just reviewed that make up the three `T3DLIB` library modules from the first *Tricks*. Remember, our goal was to implement a double-buffered graphics system that supports 8- and 16-bit windowed and full-screen graphics, and that models our virtual computer interfaces functionality. We have exactly what we need to do that now.

The next step is to create a really solid Game Console "template" based on the alpha version we made a few sections back. However, before we go there, let's take a moment to look at the functionality of the virtual graphics interface, and the exact mapping to the real graphics interface in `T3DLIB1.CPP`. The sound and music stuff is much easier to grasp, so I'm not going to show the mapping. Moreover, the demos that follow in the final sections of this chapter have numerous examples of sound, music, and input, so you can see the functions in real programs. Even so, taking a look at the graphics part is worth the work, so let's do it.

Mapping the Real Graphics to the Unreal Graphics of the Virtual Interface

Earlier in this chapter, we went through a mental experiment and came up with the most generic graphics interface that we could think of as the minimum necessary to perform software-based 3D graphics on any computer. Now, with the tools of the libraries at hand, let's take a look at what the final mappings are from virtual to real.

Starting the System

In the virtual computer's graphics interface, we assumed that there was a function called `Create_Window()` with the prototype

```
Create_Window(int width, int height, int bit_depth);
```

that would take care of all the details of getting the graphics system ready, along with opening a window of the desired size and bit depth. In our real implementation, this functionality has to be split into two function calls, because we want to try and keep windows and DirectX separate. The first function calls are standard Windows calls to create a standard Windows window.

Playrix Ex. 1023, page 239

05 0672318350 CH03 05•02•2003 2:34 PM Page 215

Now, listen up: If you are creating a full-screen application, you should use the window flags of `WM_POPUP`, but if you want a windowed application, you should use something like `WM_OVERLAPPEDWINDOW`. Hence, getting the graphics system ready is a two-step process:

1. Make a call to the Win32 API function `Create_Window()` with the appropriate parameters for either a full-screen window or a windowed application. With a full-screen window, you don't usually want any controls.
2. Call the DirectX wrapper function, along with the handle to the window (which is global) and the appropriate parameters to finish the job.

All the action really happens during step 2. This is where DirectDraw is initialized, the frame buffers are created, the palette for 8-bit modes is generated, and the clippers are attached to the graphics window and back buffer.

Because we are going to use the Game Console template, the plan of attack is to let the `WinMain()` function open the window up, and then in the call to `Game_Init()`, make a call to the function `DDraw_Init()`, which does all the dirty work. If you recall, here's its prototype:

```
int DDraw_Init(int width, int height, int bpp, int windowed=0);
```

Simple, huh? Well, just look inside the code to really appreciate all it does. Anyway, to recap, the single call in our virtual computer interface to create a window (in fantasy land) was `Create_Window()`. In real life, we need two calls to implement this functionality: one to create a Windows window, and the second to initialize DirectDraw and connect it to the window.

Global Mappings

The first things that we designed into our virtual graphics interface were two frame buffers: one visible, and one offscreen. We called these the primary and secondary (back) buffers, as shown in Figure 3.12.

Additionally, we agreed that for any given resolution and bit depth, these buffers would be linearly addressable, with the one constraint that the memory pitch to jump from line to line might not be the same as the pixel pitch, and that there would be variables to track this. The names of the virtual frame buffer pointers and the pitch variables were as follows:

```
UCHAR *primary_buffer; // the primary buffer
int  primary_pitch; // the memory pitch in bytes

UCHAR *secondary_buffer; // the secondary buffer
int  seconday_pitch;  // the memory pitch in bytes
```

Playrix Ex. 1023, page 240

05 0672318350 CH03 05•02•2003 2:34 PM Page 216

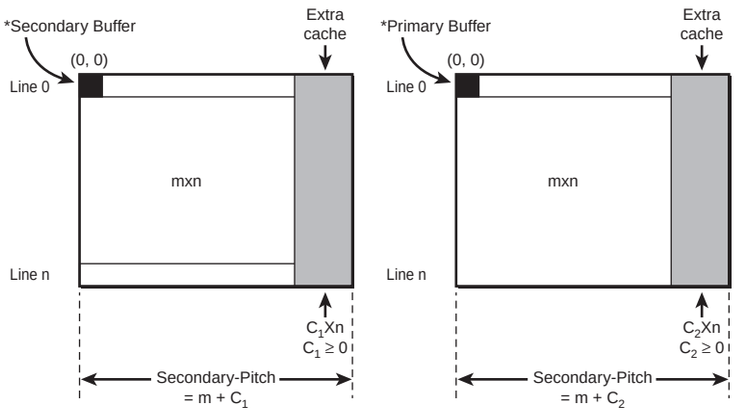


FIGURE 3.12 The frame buffers.

In the real graphics library under `T3DLIB1.CPP`, there is almost an identical mapping. The variables that serve the same purpose are as follows:

```
LPDIRECTDRAWSURFACE7 lpddsprimary;   // dd primary surface
LPDIRECTDRAWSURFACE7 lpddsback;    // dd back surface

UCHAR        *primary_buffer; // primary video buffer
UCHAR        *back_buffer;   // secondary back buffer
int         primary_lpitch;  // memory line pitch
int         back_lpitch;   // memory line pitch
```

You'll notice two extra DirectX-centric variables: `lpddsprimary` and `lpddsback`. These are DirectDraw surface pointers to both the primary and secondary (back) surfaces. They are needed for some calls, and hence you should know them. Additionally, the width and height of the primary and secondary buffers are always the same. Although the primary buffer's client area might only be a window and not the entire desktop, the secondary buffer will *always* be the same size as the client area. Bottom line—the visible primary frame buffer and the invisible secondary frame buffer are always the same size and bit depth.

256-Color Modes

Although you saw the 256-color palette manipulation functions in the graphics API listing of `T3DLIB1.CPP¦H`, I just wanted to reinforce them. The functions that enable you to manipulate the palette are as follows:

```
int Set_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename, LPPALETTEENTRY palette);
```

Playrix Ex. 1023, page 241

05 0672318350 CH03 05•02•2003 2:34 PM Page 217


```
int Save_Palette_To_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
```

I have highlighted the functions of most interest. They are used to alter a single palette entry, or the entire palette at once. In most cases, it's inefficient to change one palette entry at a time. It's better to make all the changes at once. The data structure you pass is either a pointer to a single PALETTENTRY or to an array of them. Here's the PALETTENTRY data structure to refresh your memory:

```
typedef struct tagPALETTEENTRY { // pe
  BYTE peRed;  // red channel 8-bits
  BYTE peGreen; // green channel 8-bits
  BYTE peBlue;  // blue channel 8-bits
  BYTE peFlags; // flags control, PC_EXPLICIT for the first and last 10
          // colors if windowed mode, else use PC_NOCOLLAPSE
} PALETTEENTRY;
```

Finally, if you do select a 256-color mode, the palette data in PALDATA2.PAL will be loaded. This is a palette that I like to use, and it has good color space coverage. You can take a look at it by opening PALDATA2.BMP. Of course, you can always load your own 256-color palette off disk or from a bitmap's attached palette.

Locking/Unlocking Function Mappings

Alright, everything is looking good. Now let's take a look at the four functions we made up to lock/unlock the primary and secondary surfaces. In the virtual computer interface, they looked like this:

```
Lock_Primary(UCHAR **primary_buffer, int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);

Lock_Secondary(UCHAR **secondary _buffer, int *secondary _pitch);
Unlock_Secondary(UCHAR *secondary _buffer);
```

In the real graphics library, we have these functions that implement that functionality:

```
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);

UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

Playrix Ex. 1023, page 242

05 0672318350 CH03 05•02•2003 2:34 PM Page 218

The only difference is that the real functions alter the globals

```
UCHAR *primary_buffer;   // primary video buffer
UCHAR *back_buffer;     // secondary back buffer
int  primary_lpitch;    // memory line pitch
int  back_lpitch;     // memory line pitch
```

directly, and hence don't need any parameters.

Animation Function Mapping

The last function that I want to mention the mapping for is the animation function that flips the secondary surface buffer to the primary (or copies it in some cases). If you remember, we called it `Flip_Display()` in the virtual computer software interface model. This function is what makes smooth animation possible. In the real library, the function that does the job is

```
int DDraw_Flip(void);
```

Nothing more than a different name, really. The functionality of the desired virtual model is exactly implemented in this function. `DDraw_Flip()` will work perfectly for both full-screen and windowed applications.

Well, that's it for the function mapping of the virtual computer model with our real functions. I just wanted you to really understand what we are trying to do—create a generic graphics system with an API that we have handy that abstracts the graphics rendering enough that we only think in terms of two frame buffers and a few functions. I really hate dumping an API on you, but my hands are tied (all eight of them), and I had to use something to implement the virtual computer interface for which we had the source.

The Final `T3DLIB` Game Console

Now we're finally ready to see the complete Game Console that we are going to use to create all of the demos and game programs from here on out. Of course, we might add to it, but for the most part what you are about to see captures all the functionality we need. The upgraded Game Console is named `T3DCONSOLE2.CPP`. Its source is shown in the following.

NOTE

You might be wondering what happened to `T3DCONSOLE.CPP¦EXE`. In the first *Tricks*, I created `T3DCONSOLE.CPP¦EXE`, so it only seems appropriate to append a "2" onto the new version, because this book is really a continuation of the previous *Tricks* and we want to keep our naming conventions consistent.

```
// T3DCONSOLE2.CPP - First template for Tricks 3D Vol II
// Use this as a template for your applications if you wish
// you may want to change things like the resolution of the
```

Playrix Ex. 1023, page 243

05 0672318350 CH03 05•02•2003 2:34 PM Page 219


```
// application, if it's windowed, the directinput devices
// that are acquired and so forth...
// currently the app creates a 640x480x16 windowed display
// hence, you must be in 16 bit color before running the application
// if you want fullscreen mode then simple change the WINDOWED_APP
// value in the #defines below value to FALSE (0). Similarly, if
// you want another bitdepth, maybe 8-bit for 256 colors then
// change that in the call to DDraw_Init() in the function
// Game_Init() within this file.


// READ THIS!
// To compile make sure to include DDRAW.LIB, DSOUND.LIB, DINPUT.LIB,
// DINPUT8.LIB, WINMM.LIB in the project link list, and of course
// the C++ source modules T3DLIB1.CPP,T3DLIB2.CPP, and T3DLIB3.CPP
// and the headers T3DLIB1.H,T3DLIB2.H, and T3DLIB3.H must
// be in the working directory of the compiler

// INCLUDES ///////////////////////////////////////////////

#define INITGUID    // make sure al the COM interfaces are available
            // instead of this you can include the .LIB file
            // DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h>  // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h> // directX includes
#include <dsound.h>
```

Playrix Ex. 1023, page 244

05 0672318350 CH03 05•02•2003 2:34 PM Page 220


```
#include <dmksctrl.h>
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h" // game library includes
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// DEFINES ////////////////////////////////////////////////

// defines for windows interface
#define WINDOW_CLASS_NAME "WIN3DCLASS" // class name
#define WINDOW_TITLE   "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH   640  // size of window
#define WINDOW_HEIGHT   480

#define WINDOW_BPP    16  // bitdepth of window (8,16,24 etc.)
                // note: if windowed and not
                // fullscreen then bitdepth must
                // be same as system bitdepth
                // also if 8-bit the a pallete
                // is created and attached

#define WINDOWED_APP   1   // 0 not windowed, 1 windowed

// PROTOTYPES /////////////////////////////////////////////

// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// GLOBALS ////////////////////////////////////////////////

HWND main_window_handle      = NULL; // save the window handle
HINSTANCE main_instance      = NULL; // save the instance
char buffer[256];             // used to print text



// FUNCTIONS //////////////////////////////////////////////
```

Playrix Ex. 1023, page 245

05 0672318350 CH03 05•02•2003 2:34 PM Page 221


```
LRESULT CALLBACK WindowProc(HWND hwnd,
              UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT  ps;    // used in WM_PAINT
HDC    hdc;    // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
  // do initialization stuff here
  return(0);
  } break;

  case WM_PAINT:
     {
     // start painting
     hdc = BeginPaint(hwnd,&ps);

     // end painting
     EndPaint(hwnd,&ps);
     return(0);
    } break;

  case WM_DESTROY:
    {
    // kill the application
    PostQuitMessage(0);
    return(0);
    } break;

  default:break;

  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

Playrix Ex. 1023, page 246

05 0672318350 CH03 05•02•2003 2:34 PM Page 222


```
// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain(  HINSTANCE hinstance,
      HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow)
{
// this is the winmain function

WNDCLASS winclass; // this will hold the class we create
HWND   hwnd;     // generic window handle
MSG   msg;     // generic message
HDC   hdc;    // generic dc
PAINTSTRUCT ps;   // generic paintstruct

// first fill in the window class stucture
winclass.style  = CS_DBLCLKS ¦ CS_OWNDC ¦ CS_HREDRAW ¦ CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra  = 0;
winclass.cbWndExtra  = 0;
winclass.hInstance  = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;

// register the window class
if (!RegisterClass(&winclass))
  return(0);

// create the window, note the test to see if WINDOWED_APP is
// true to select the appropriate window flags
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
       WINDOW_TITLE,   // title
       (WINDOWED_APP ? (WS_OVERLAPPED ¦ WS_SYSMENU ¦ WS_CAPTION) :
            (WS_POPUP ¦ WS_VISIBLE)),
        0,0,    // x,y
        WINDOW_WIDTH, // width
            WINDOW_HEIGHT, // height
        NULL,    // handle to parent
       NULL,    // handle to menu
       hinstance,// instance
```

Playrix Ex. 1023, page 247

05 0672318350 CH03 05•02•2003 2:34 PM Page 223


```
      NULL)))  // creation parms
return(0);

// save the window handle and instance in a global
main_window_handle = hwnd;
main_instance   = hinstance;

// resize the window so that client is really width x height
if (WINDOWED_APP)
{
// now resize the window, so the client area is the actual size requested
// since there may be borders
// alsocontrols if this is going to be a windowed app
// if the app is not windowed then it won't matter
RECT window_rect = {0,0,WINDOW_WIDTH-1,WINDOW_HEIGHT-1};

// make the call to adjust window_rect
AdjustWindowRectEx(&window_rect,
   GetWindowStyle(main_window_handle),
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// save the global client offsets, they are needed in DDraw_Flip()
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// now resize the window with a call to MoveWindow()
MoveWindow(main_window_handle,
      0, // x position
      0, // y position
      window_rect.right - window_rect.left, // width
      window_rect.bottom - window_rect.top, // height
      FALSE);

// show the window, so there's no garbage on first render
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed

// perform all game console specific initialization
Game_Init();

// disable CTRL-ALT_DEL, ALT_TAB, comment this line out
// if it causes your system to crash
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);
```

Playrix Ex. 1023, page 248

05 0672318350 CH03 05•02•2003 2:34 PM Page 224


```
// enter main event loop
while(1)
   {
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
  // test if this is a quit
    if (msg.message == WM_QUIT)
      break;

  // translate any accelerator keys
  TranslateMessage(&msg);

  // send the message to the window proc
  DispatchMessage(&msg);
  } // end if

    // main game processing goes here
    Game_Main();

   } // end while

// shutdown game and release all resources
Game_Shutdown();

// enable CTRL-ALT_DEL, ALT_TAB, comment this line out
// if it causes your system to crash
SystemParametersInfo(SPI_SCREENSAVERRUNNING, FALSE, NULL, 0);

// return to Windows like this
return(msg.wParam);

} // end WinMain

// T3D II GAME PROGRAMMING CONSOLE FUNCTIONS ////////////////

int Game_Init(void *parms)
{
// this function is where you do all the initialization
// for your game

// start up DirectDraw (replace the parms as you desire)
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);
```

Playrix Ex. 1023, page 249

05 0672318350 CH03 05•02•2003 2:34 PM Page 225


```
// initialize directinput
DInput_Init();

// acquire the keyboard
DInput_Init_Keyboard();

// add calls to acquire other directinput devices here...


// initialize directsound and directmusic
DSound_Init();
DMusic_Init();

// hide the mouse
ShowCursor(FALSE);

// seed random number generator
srand(Start_Clock());

// all your initialization code goes here...


// return success
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// this function is where you shutdown your game and
// release all resources that you allocated

// shut everything down

// release all your resources created for the game here....


// now directsound
DSound_Stop_All_Sounds();
DSound_Delete_All_Sounds();
DSound_Shutdown();
```

Playrix Ex. 1023, page 250

05 0672318350 CH03 05•02•2003 2:34 PM Page 226


```
// directmusic
DMusic_Delete_All_MIDI();
DMusic_Shutdown();

// release all input devices
DInput_Release_Keyboard();

// shut down directinput
DInput_Shutdown();

// shutdown directdraw last
DDraw_Shutdown();

// return success
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for you game go here!

int index; // looping var

// start the timing clock
Start_Clock();

// clear the drawing surface
DDraw_Fill_Surface(lpddsback, 0);

// read keyboard and other devices here
DInput_Read_Keyboard();

// game logic here...



// flip the surfaces
DDraw_Flip();
```

Playrix Ex. 1023, page 251

05 0672318350 CH03 05•02•2003 2:34 PM Page 227


```
// sync to 30ish fps
Wait_Clock(30);

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) ¦¦ keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if

// return success
return(1);

} // end Game_Main
```


Let's take a look at some of the key points of the Game Console, and then we'll move on to using it. First, you'll notice that some of the code is highlighted in bold to indicate important code sections. Let's take a look at what does what.

Opening the Game Console Window

The first lines of code that are highlighted are in the `#defines` section:

```
#define WINDOW_WIDTH   640  // size of window
#define WINDOW_HEIGHT   480

#define WINDOW_BPP    16  // bitdepth of window (8,16,24 etc.)
                // note: if windowed and not
                // fullscreen then bitdepth must
                // be same as system bitdepth
                // also if 8-bit the a palette
                // is created and attached

#define WINDOWED_APP   1   // 0 not windowed, 1 windowed
```

These are very important because they control the size of the window (or full-screen size), the bit depth, and whether you want a windowed display rather than full screen. Currently, I have them set to 640×480 for the window size, 16-bits per pixel, and a windowed display. These parameters are used in a number of places in the code. However, the two most important are when the window is created in `WinMain()` along with the call to `DDraw_Init()` in `Game_Main()`. Let's talk about the call in `WinMain()` first.

If you take a look at the `Create_Window()` call in `WinMain()` (copied in the following fragment for your convenience), you will notice that there is a ternary conditional operator that tests whether the application is windowed.

Playrix Ex. 1023, page 252

05 0672318350 CH03 05•02•2003 2:34 PM Page 228


```
// create the window, note the test to see if WINDOWED_APP is
// true to select the appropriate window flags
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
       WINDOW_TITLE,   // title
       (WINDOWED_APP ? (WS_OVERLAPPED ¦ WS_SYSMENU ¦ WS_CAPTION) :
            (WS_POPUP ¦ WS_VISIBLE)),
        0,0,    // x,y
        WINDOW_WIDTH, // width
             WINDOW_HEIGHT, // height
        NULL,    // handle to parent
       NULL,    // handle to menu
       hinstance,// instance
       NULL)))  // creation parms
return(0);
```

This is so the window is created with the proper Windows flag. If we want a windowed display, we want to see the window borders and have a couple of controls such as a close box, so the Window flags (`WM_OVERLAPPED ¦ WM_SYSMENU ¦WS_CAPTION`) are used.

On the other hand, if the code selects a full-screen mode, the window is made the same size as the DirectDraw surface, but with no controls; hence the window style `WM_POPUP` is used. Additionally, once the window is created (either in preparation for a windowed application or full-screen) the next section of code is rather tricky. It basically resizes the window so that the client area is the size that we are requesting, rather than the size we are requesting minus the size of the border and controls.

If you recall from Windows programming, when you create a window with size `WINDOW_WIDTH x WINDOW_HEIGHT`, it doesn't mean the client area will be `WINDOW_WIDTH x WINDOW_HEIGHT`. It means the window's entire area will be `WINDOW_WIDTH x WINDOW_HEIGHT`. Therefore, if there are no controls or borders, the client area *will be* `WINDOW_WIDTH x WINDOW_HEIGHT`, but if there are controls, we lose a little, as shown in Figure 3.13.

To solve this problem, we need to resize the window on the fly so that the size of the client area is the exact size that we are requesting for the window. Here's the code that does this:

```
// resize the window so that client is really width x height
if (WINDOWED_APP)
{
// now resize the window, so the client area is the actual size requested
// since there may be borders and controls if this is going to be a windowed app
// if the app is not windowed then it won't matter
RECT window_rect = {0,0,WINDOW_WIDTH-1,WINDOW_HEIGHT-1};

// make the call to adjust window_rect
AdjustWindowRectEx(&window_rect,
```

Playrix Ex. 1023, page 253

05 0672318350 CH03 05•02•2003 2:34 PM Page 229


```
   GetWindowStyle(main_window_handle),
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// save the global client offsets, they are needed in DDraw_Flip()
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// now resize the window with a call to MoveWindow()
MoveWindow(main_window_handle,
      0, // x position
      0, // y position
      window_rect.right - window_rect.left, // width
      window_rect.bottom - window_rect.top, // height
      FALSE);

// show the window, so there's no garbage on first render
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed
```

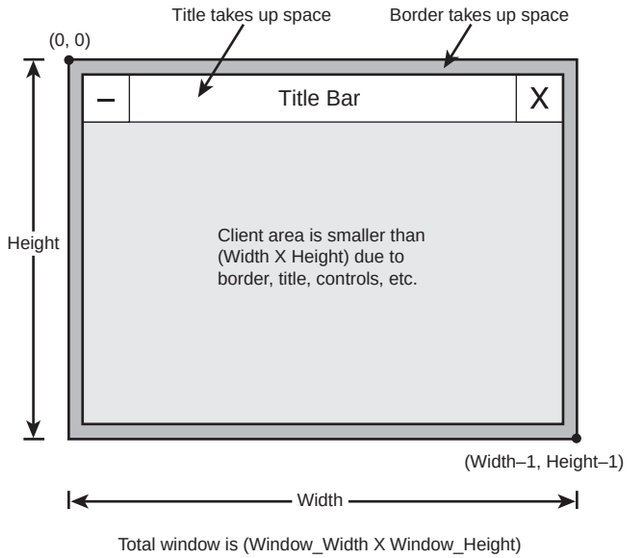


FIGURE 3.13 Total window area compared to client area.

There are a couple of other ways to do this, but this is the one I usually use. Another way is to know what controls you have, query Windows for the size of the controls, and then compute the size the window should be with the controls. Whatever way you do it, just

Playrix Ex. 1023, page 254

05 0672318350 CH03 05•02•2003 2:34 PM Page 230

make sure that if a windowed app is selected, the client area is exactly `WINDOW_WIDTH` x `WINDOW_HEIGHT`.

> **NOTE**
>
> If you set `WINDOWED_APP` to `0`, this logic isn't executed, because a `WM_POPUP` window has no controls and we don't need to resize.

After the window is created and resized (if needed), at some point a call to `Game_Init()` is made. Within it you will add all your logic and initialization in the future. However, at this point, it has the bare minimum to get things going. In line with our discussion, a call to `DDraw_Init()` is made:

```
// start up DirectDraw (replace the parms as you desire)
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);
```

You will notice that you don't have to do anything. Just make sure to set the `#defines` up at the top of the file to what you want and whammo—you're Kid Rock!

> **NOTE**
>
> You might notice those calls to `SystemParametersInfo()` around the main event loop. These calls make Windows think that the screensaver is on, and to ignore ALT+TAB. DirectX applications can get really mangled if you don't handle ALT+TAB-ing correctly (which we don't), so this way you hopefully don't have to worry about it. If you're interested, take a look at the DirectX SDK, but in general when your application loses focus, you have to restore all lost surfaces and reacquire all input devices and so on. It's a real pain!

Using and Compiling the Game Console

Basically `T3DCONSOLE2.CPP` is just a "shell" or template that you will use to create applications. You can use it to handle all the Windows stuff, and then just drop your code into the game functions `Game_Init()`, `Game_Main()`, and `Game_Shutdown()`. However, to compile it, you still need to include the following files:

- `T3DLIB1.CPP¦H`—The DirectDraw module
- `T3DLIB2.CPP¦H`—The DirectInput module
- `T3DLIB3.CPP¦H`—The DirectSound and DirectMusic module

These files must be in your root:

- `PALDATA1¦2.PAL`—The default palettes for 256-color mode

And you must, of course, link with

- `DDRAW.LIB`, `DSOUND.LIB`, `DINPUT.LIB`, and `DINPUT8.LIB`

Playrix Ex. 1023, page 255

05 0672318350 CH03 05•02•2003 2:34 PM Page 231

Please set the compiler target to `Win32 .EXE application`. Don't forget to add the DirectX .LIB files to the link list, and to set the search paths for the compiler to find all the headers.

Just for fun, I have compiled the Game Console without anything in it, creating the `T3DCONSOLE2.EXE`. The application doesn't do much, but creates a 640×480×16-bit windowed display. However, to run it you must have your desktop in 16-bit mode.

CAUTION

Windowed applications based on the Game Console will *not* change the bit depth of the desktop. You can alter it to do this if you want, but it can be catastrophic. Here's why: If you have other applications running and you then start up a DirectDraw application that changes the bit depth of the screen, but creates a windowed display rather than a full-screen display, the other application might function improperly and/or crash the system if you switch focus to it. As a rule of thumb for full-screen applications, do what you will with resolution and bit depth. But for windowed applications, you might want to test for the bit depth and see whether it meets your target before starting your application and letting the user switch the desktop.

I could have just as easily made the application a full-screen 640×480×16 application with a single change to the `#defines`:

```
#define WINDOWED_APP   0   // 0 not windowed, 1 windowed
```

However, windowed applications are more fun and a little easier to debug, so I am supporting them heavily in this book. Nevertheless, when your game is finally ready to run, I suggest dumping the system into full-screen mode.

Sample T3LIB Applications

We are almost done with this chapter—whoooweee! I'm getting sick of this stuff—I'm dying to get to the 3D part! Anyway, just to be complete, what I want to do now is show you a number of sample applications that use the Game Console `T3DCONSOLE2.CPP` as the starting point for real demos that show off graphics, sound, music, and input. The demos are going to actually *do something*, because I am hijacking them from the first *Tricks*. However, they are all updated now. This way, you can see some real examples of using the console template. However, note that the sound and music demos are less glamorous, and don't really need much.

Lastly, as we proceed through the book and build the `T3DLIB` game engine, we will add modules `T3DLIB4`, `5`, `6`, and so forth each chapter or so.

Windowed Applications

As it stands, the `T3DCONSOLE2.CPP` is really a demonstration of windowed 16-bit applications, but I always like to see something with a little more girth to it—don't you?

Playrix Ex. 1023, page 256

05 0672318350 CH03 05•02•2003 2:34 PM Page 232

Therefore, I converted one of the artificial intelligence demos from the first *Tricks* (pattern systems demo) to 16-bit and put it in a window. The name of the program on the CD is `DEMOII3_1.CPP¦EXE` and a screenshot is shown in Figure 3.14. To run the program, make sure the desktop is in 16-bit color mode. Also, if you want to compile it, make sure you include all the library modules!



FIGURE 3.14 A screenshot of a windowed application AI demo.

In any case, the demo not only shows off 16-bit windowed mode, but it shows how to load bitmaps, use the BOB library, make sounds, and a lot more all in 16-bit graphics mode. Lots to learn by studying the code.

Full-Screen Applications

Writing a full-screen application is no different with our library; in fact, it's better because there is less chance of resource sharing problems. Moreover, in 256-color modes, you have full control over the entire palette, rather than losing access to the first and last ten entries, as with windowed 256-color modes. Additionally, you can switch into any bit depth and resolution with full-screen modes and not worry about the desktop.

As a demo of a full-screen application, I have taken one of the 256-color physics demos from the first *Tricks* and worked it into our new Game Console template. The name of the program is `DEMOII3_2.CPP¦EXE`. Figure 3.15 shows a screenshot of the program in action. This program again illustrates the use of bitmaps, BOBs, and palettized 256-color modes.

Playrix Ex. 1023, page 257

05 0672318350 CH03 05•02•2003 2:34 PM Page 233

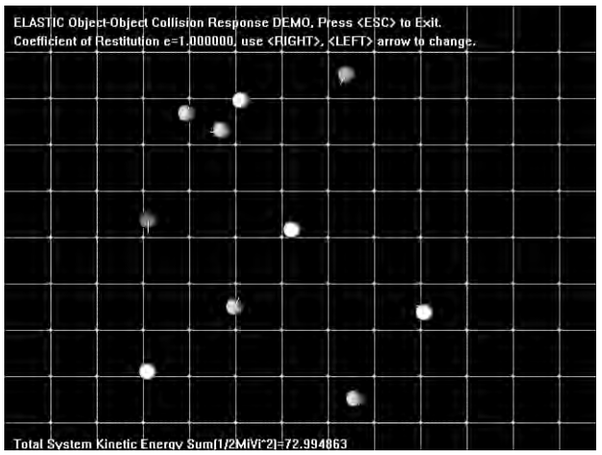


FIGURE 3.15 A screenshot of a 256-color, full-screen physics demo.

Sound and Music

Working with sound and music under DirectSound and DirectMusic isn't hard, but it's not easy, either. The systems are very complex, and have a lot of functionality. However, all we want to be able to do is load sound and music off disk and play the darn things! The `T3DLIB2.CPP¦H` module handles all this in a very clean and simple way.

With that in mind, the sound and music demo is going to be rather plain graphically, so you can focus on just the calls to make sound and music work. Therefore, I have taken a demo from the first *Tricks* that basically works with both DirectSound and DirectMusic at the same time. The demo is a simple Windows application that has a menu and enables you to select a MIDI song to play, while at the same time you can "fire" off sound effects. Also, the demo structure is based on a stripped down version of the Game Console to minimize anything extraneous.

The name of the demo is `DEMOII3_3.CPP¦EXE`, and Figure 3.16 features a screenshot of the application in action (what there is of it). Additionally, this demo uses a couple of resources such as a menu and cursor; thus, there are a few extra files that you need to add to your project:

- `DEMOII3_3.RC`—The Windows resource file with the menu
- `DEMOII3_3RES.H`—The header with identifiers for the resources
- `T3DX.ICO`—The cursor icon used by the program

Playrix Ex. 1023, page 258

05 0672318350 CH03 05•02•2003 2:34 PM Page 234



FIGURE 3.16 A screenshot of the sound and music demo.

As usual, this is still a DirectX application, and you need to include the DirectX .LIB files. Technically, you don't need `DDRAW.LIB`, but you should always keep all of them in your link list to be safe. Also, because there aren't any graphics or input calls to `T3DLIB1.CPP` or `T3DLIB2.CPP`, do *not* include the .CPP files in the build—you will get link errors if you do!

Working with Input

The last examples of using the library are input demos. As you know, there are three primary input devices used in a game:

- Keyboard
- Mouse
- Joystick

Of course, these days a joystick means a lot of things, but usually we can classify all devices that aren't a mouse or keyboard as a joystick.

Using the input library is very simple. It's contained in `T3DLIB2.CPP¦H` and supports the keyboard, mouse, and joystick devices. With just a few function calls, you can initialize DirectInput, acquire an input device, and read from it in your main loop. What we are going to do is review demos of each device: keyboard, mouse, and joystick. Of course, you are more than free to use all input devices together, but the demos will keep them separate, so you can better see what's going on.

Keyboard Demo

As usual, the demo are based on code from the first *Tricks*. Let's begin with the keyboard demo `DEMOII3_4.CPP¦EXE`. A screenshot is shown in Figure 3.17. Basically, this demo enables you to move a BOB skeleton around the screen. If you look at the code that reads the keyboard, you will notice that the data structure returned is nothing more than a 256 `BYTE` array of `BOOLEAN`s, each representing one key. The table is shown here:

```
UCHAR keyboard_state[256]; // contains keyboard state table
```

Playrix Ex. 1023, page 259

05 0672318350 CH03 05•02•2003 2:34 PM Page 235

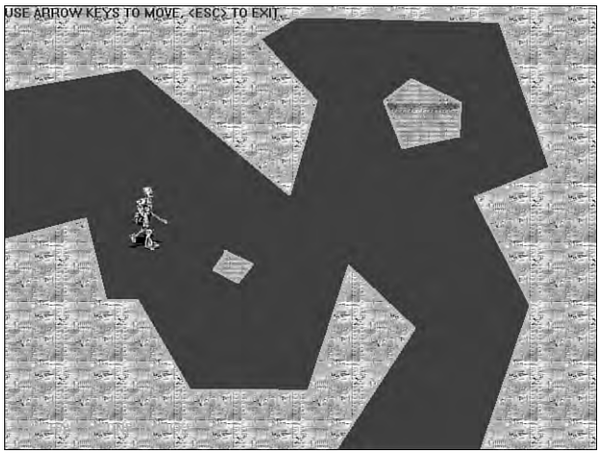


FIGURE 3.17 A screenshot of the keyboard demo.

To access the table, you use the DirectInput keyboard constants, which are all of the form `DK_*`. Table 3.2 from a previous section lists some of the more common constants. All you need to do to test whether a key is pressed. After you have retrieved the keyboard data with a call to `DInput_Read_Keyboard()`, you can use the following simple test:

```
if (keyboard_state[DIK_UP])
  {
  // if the up arrow is pressed this will execute
  } // end if
```

To compile the demo program, you will need all the DirectX .LIB files, along with `T3DLIB1.CPP¦H`, `T3DLIB2.CPP¦H`, and `T3DLIB3.CPP¦H`.

Mouse Demo

The mouse demo is again based on code from the first *Tricks*. It's a little paint program, as shown in Figure 3.18. The code is in the file `DEMOII3_5.CPP¦EXE`. The program shows off a number of interesting things about GUI programming, such as object picking, mouse tracking, and so forth. So, make sure to check out the painting code in addition to the mouse tracking code.

The mouse input device can operate in two modes: absolute or relative. I prefer relative because I can always figure out where I am, so the API wrapper puts the system in that mode. Hence, each time you read the mouse, you get deltas from the last call. The mouse is read each time you call `DInput_Read_Mouse()` and is stored in the following global mouse record:

```
DIMOUSESTATE mouse_state; // contains state of mouse
```

Playrix Ex. 1023, page 260

05 0672318350 CH03 05•02•2003 2:34 PM Page 236

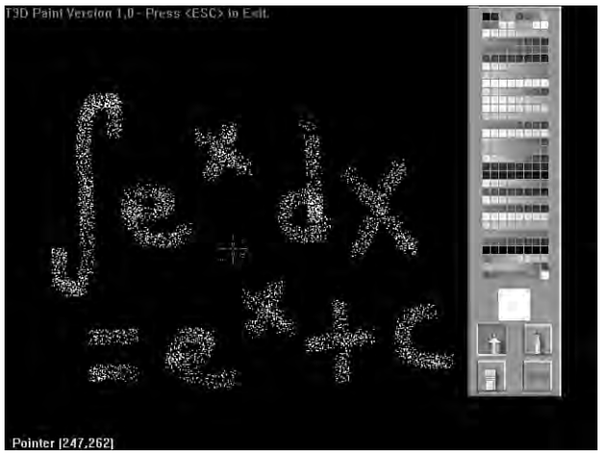


FIGURE 3.18 A screenshot of the mouse-based painting program.

Once again, the structure of a `DIMOUSESTATE` object is as follows:

```
typedef struct DIMOUSESTATE {
  LONG lX; // x-axis
  LONG lY; // y-axis
  LONG lZ; // z-axis
  BYTE rgbButtons[4]; // state of the buttons
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

To compile this program, you will need all the DirectX .LIB files, along with `T3DLIB1.CPP¦H`, `T3DLIB2.CPP¦H`, and `T3DLIB3.CPP¦H`.

Joystick Demo

The "joystick" device on computers these days can be almost anything from gamepads to paddles. However, any joystick-like device is nothing more than a collection of two types of input devices:

- Analog axes
- Buttons

For example, if you take a look at Figure 3.19, we see four different joystick-like devices: a typical joystick, an advanced joystick, a game pad, and a steering wheel/paddle. You might hesitate to call these all joysticks, but they really are. The joystick has two axes: the x and y, along with two buttons. The gamepad is all buttons. And finally, the paddle consists of a rotational axis along with a couple of buttons.

Playrix Ex. 1023, page 261

05 0672318350 CH03 05•02•2003 2:34 PM Page 237



FIGURE 3.19 Four basic joystick devices.

Of course, in DirectInput speak, there are constants to define some of the most prevalent subtypes of joysticks, like plain joysticks, flight sticks, paddles, and gamepads, but in general they are all used in the same way. That is, you retrieve either smoothly varying data back from an analog slider or yoke, and a number of digital on/off signals representing the buttons. The data record returned the call to `DInput_Read_Joystick()` from the DirectInput module of our library is stored in the global

```
DIJOYSTATE joy_state;   // contains state of joystick
```

which is of the form

```
typedef struct DIJOYSTATE {
  LONG  lX; // x-axis
  LONG  lY; // y-axis
  LONG  lZ; // z-axis
  LONG  lRx; // rotation about x-axis
  LONG  lRy; // rotation about y-axis
  LONG  lRz; // rotation about z-axis
  LONG  rglSlider[2]; // u,v slider positions
  DWORD  rgdwPOV[4];  // point of view hat state
  BYTE  rgbButtons[32]; // state of buttons 0..31
} DIJOYSTATE, *LPDIJOYSTATE;
```

As a demo of using the joystick device, I have hijacked yet another program from the first *Tricks* that is a little *Centipede*-like game (not much of a game, though). The program is named `DEMOII3_6.CPP¦EXE` on the CD. A screenshot is shown in Figure 3.20. As usual, you will need all the DirectX .LIB files, along with `T3DLIB1.CPP¦H`, `T3DLIB2.CPP¦H`, and `T3DLIB3.CPP¦H` to compile.

NOTE

Make sure you check out some of the aspects of the demo relevant to game programming, such as the primitive game logic, collision detection, and weapon systems.

Playrix Ex. 1023, page 262

05 0672318350 CH03 05•02•2003 2:34 PM Page 238

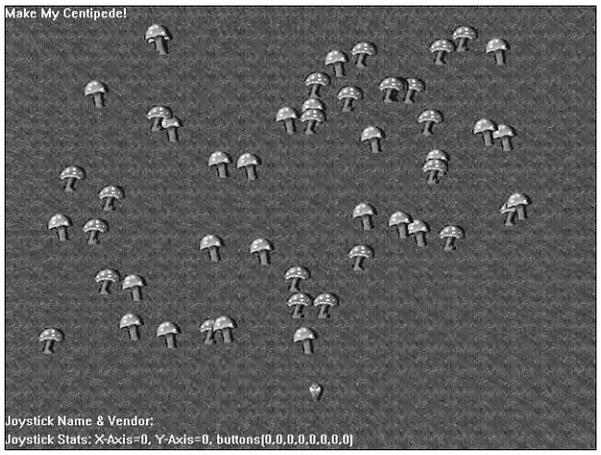


FIGURE 3.20 A screenshot of the joystick demo.

Summary

This chapter has really been a "transition" from the first *Tricks* to this book. You should understand that 3D graphics isn't about DirectX or Windows; it's about math, math, and more math. And all we need to experiment with 3D graphics is a platform that has a few functions and a couple of frame buffers. Also, even if you haven't read the first *Tricks*, you should have a decent handle on the API that we are going to use throughout the book. Either way, you should feel confident because we're only going to use a few functions to implement our virtual graphics computer needed for the rest of the text. For those of you that have read the first *Tricks*, I bet that you are happy that the `T3DLIB` now has 16-bit windowed support! Me too. It's cool <GRIN>.

Playrix Ex. 1023, page 263

06 0672318350 Part2 05•02•2003 2:35 PM Page 239

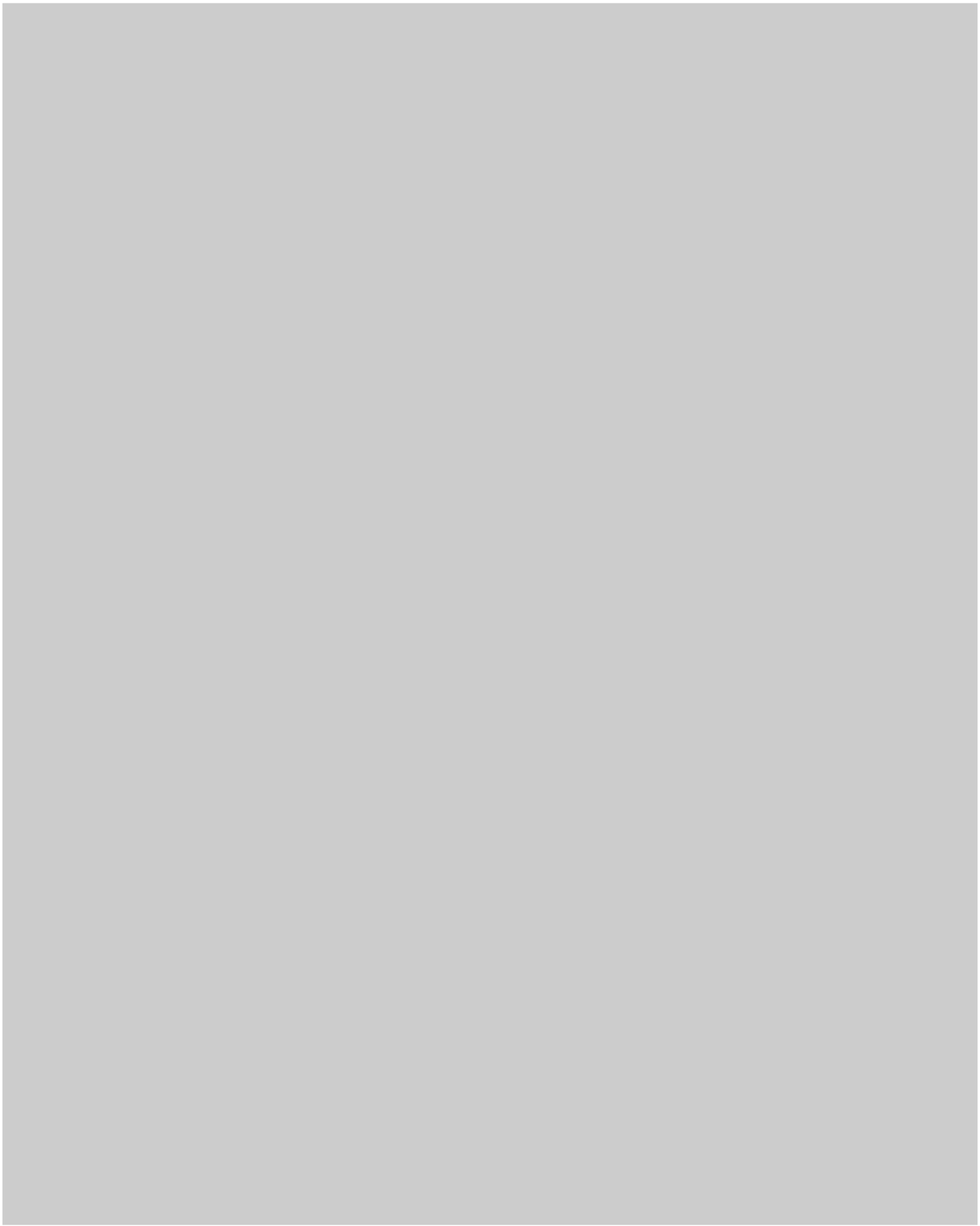
PART II

3D Math and Transformation

IN THIS PART

CHAPTER 4	It's a Math, Math, Math World—Trigonometry, Vectors, Matrices, and Quaternions	**241**
CHAPTER 5	Building a Math Engine	**363**
CHAPTER 6	Introduction to 3D Graphics	**493**
CHAPTER 7	Rendering Wireframe Worlds in 3D	**621**

Playrix Ex. 1023, page 264



07 0672318350 CH04 05•02•2003 2:35 PM Page 241

CHAPTER **4**

It's a Math, Math, Math World—Trigonometry, Vectors, Matrices, and Quaternions

"For the last time—It's Bigboo-TAY'! Tay, Tay!!!" John Bigboote'!!!"

—John Bigboote', *Buckaroo Banzai*

It used to be that you could write computer games with a fleeting knowledge of trig and a bit of algebra. Things have definitely changed these days. I remember trying to get the editor of *Game Developer Magazine* to let me write an article about vectors just a few years ago, and his reply was that the math was too advanced. Now I routinely pick the magazine up, and it's full of advanced vector calculus! With that in mind, this chapter is going to be a refresher course (for most) on some of the basic mathematics you need for 3D computer graphics. I can't teach four years of college math in a single chapter, but I can at least jar your memory, and you can always pick up a real math book if you want more. Of course, if you are a math whiz, just skip to the next chapter where we start building the math library. Anyway, here's what we'll cover:

- Mathematical notation
- Coordinate systems
- Trigonometry
- Vectors
- Matrices and linear algebra
- Basic geometric entities

IN THIS CHAPTER

- Mathematical Notation
- 2D Coordinate Systems
- 3D Coordinate Systems
- 3D Cylindrical Coordinates
- 3D Spherical Coordinates
- Trigonometry
- Vectors
- Matrices and Linear Algebra
- Basic Geometric Entities
- Using Parametric Equations
- Introduction to Quaternions
- Basic Calculus

Playrix Ex. 1023, page 266

07 0672318350 CH04 05•02•2003 2:35 PM Page 242

- Using parametric equations
- Introduction to quaternions
- Basic calculus

Mathematical Notation

Mathematics is like any language. It consists of a set of symbols that are used to represent mathematical concepts, formulas, equations, and operations. The problem with higher mathematics is that much of the symbology is literally Greek! So before we start, I want to show you some tables of commonly used symbols and conventions that I am going to try and stick to throughout the book when I am working with pure math. Although some of the concepts might be meaningless at this point, at least you have a "map" of definitions early on, rather than later in the chapter. Let's begin with Table 4.1, which contains the basic types of math objects we will work with.

TABLE 4.1 Mathematical Object Types

Type	Notation	Example
Scalar	Lower-case	a, b, x, y_1
Infinity	∞	+∞, -∞
Angle	Lower-case Greek	θ, ϕ, α, β
Vector	Lower-case bold	**u**, **v**, $\mathbf{a}_x$, $\mathbf{a}_y$
Quaternion	Lower-case bold	**p**, **q**, $\mathbf{r}_x$, $\mathbf{s}_y$
Elements of Vector	<components>	<ux, uy, uz>
Row Vector	[components]	[ux, uy, uz]
Point	Lower-case bold	**a**, $\mathbf{p}_1$, $\mathbf{p}_2$
Matrix	Upper-case bold	**A**, **B**, $\mathbf{M}_2$
Triangle	Δ followed by points	$\Delta\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$, Δ**abc**

Note: All types can be subscripted or superscripted.

Now that we have a "general" guideline for types (although I might get sloppy from time to time based on context), let's take a look at some standard mathematical operators, as shown in Table 4.2.

TABLE 4.2 Mathematical Operators

Operation	Notation	Example
Scalar Multiplication	x,* Lower-case	3x5, a*b
Matrix Multiplication	x, Lower-case	**A**x**B**
Dot Product	.	**u** . **v**
Summation	∑ expr	

Playrix Ex. 1023, page 267

07 0672318350 CH04 05•02•2003 2:35 PM Page 243

TABLE 4.2 Continued

Operation	Notation	Example
Absolute Value	|expr|	|-35|
Length	|expr|	|**u**|
Determinate	det(expr)	det(**M**)
Transpose	Superscript T	$\mathbf{v}^T$
Integration	∫ expr	
Differentiation	Prime Symbol '	x', x''

Note: "expr" stands for general expression.

Of course, we might make up some more notation as we go along, but Tables 4.1 and 4.2 are relatively decent conventions that most math books follow.

CAUTION

Generally, most mathematics books denote the determinate of matrix **M** by |**M**|, and length or norm of vector **v** by ||**v**||. The double vertical bars are too hard to lay out for general trade books like this, so we are going to use the symbology det(**M**) for determinate and |**v**| for the length and norm of a vector to simplify things, so keep that in mind. I think I am stressing the editors out enough with the Greek symbols!

2D Coordinate Systems

Just as we created some conventions for mathematical notation, we must make sure we are on the same page geometrically. Let's begin with describing and understanding the general 2D coordinate systems and then move on to the more complex 3D coordinate systems.

2D Cartesian Coordinates

The most common 2D coordinates that everyone has worked with are called *Cartesian coordinates*. Cartesian coordinates are based on a pair of perpendicular axes labeled x and y, as shown in Figure 4.1. The positive x-axis is to the right, and the negative x-axis is to the left. Conversely, the positive y-axis is in the upward direction, and the negative y-axis is in the downward direction. The point where both x = 0 and y = 0 is called the *origin*. Also, if you want to get technical, the y-axis is also called the *ordinate*, and the x-axis is the *abscissa*.

Additionally, there are four quadrants in the 2D x-y coordinate system labeled QI, QII, QIII, and QIV. The quadrants separate the four spaces by the sign on the x-y axis in that particular space. For example, Table 4.3 shows the signs of x-y in each quadrant.

Playrix Ex. 1023, page 268

07 0672318350 CH04 05•02•2003 2:35 PM Page 244

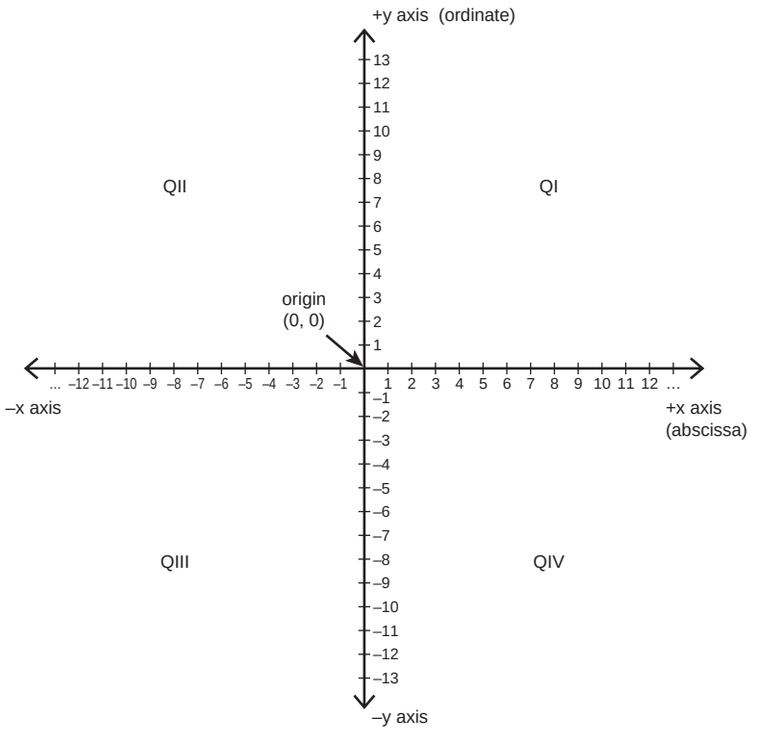


FIGURE 4.1 The Cartesian coordinate system.

TABLE 4.3 Signs in Each Quadrant

Quadrant	Sign (x)	Sign (y)
I	+	+
II	-	+
III	-	-
IV	+	-

TIP

Many graphics algorithms are optimized by solving the problem only for QI and then reflecting the solution based on symmetry to the other quadrants (or octants in a 3D system).

Finally, to locate any point in a 2D Cartesian coordinate system, you need both the x and y components. For example, the point **p**(5,3) means that x = 5 and y = 3, as shown in Figure 4.2. Easy enough? Okay, tough guy, let's move on then.

Playrix Ex. 1023, page 269

07 0672318350 CH04 05•02•2003 2:35 PM Page 245

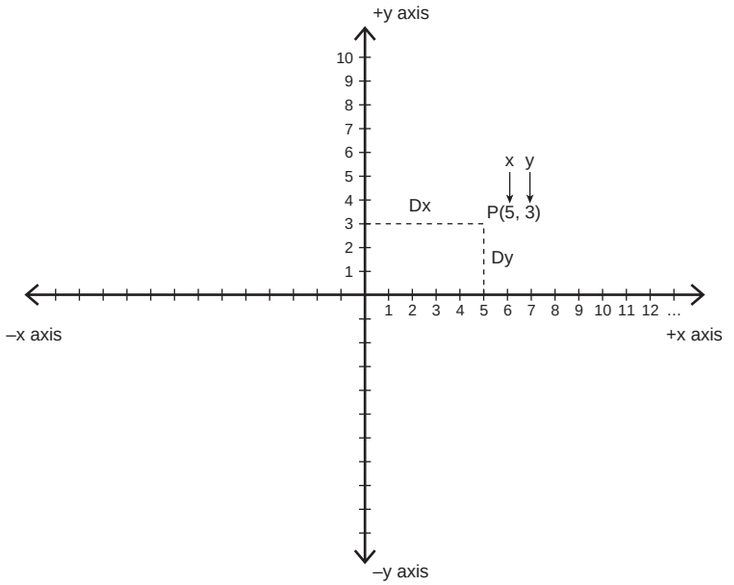


FIGURE 4.2 Locating points in the Cartesian coordinate system.

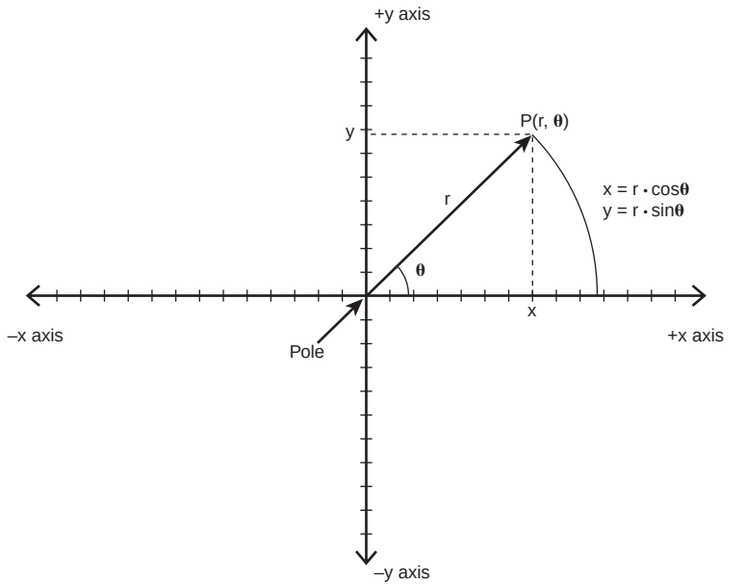


FIGURE 4.3 The 2D polar coordinate system.

Playrix Ex. 1023, page 270

07 0672318350 CH04 05•02•2003 2:35 PM Page 246

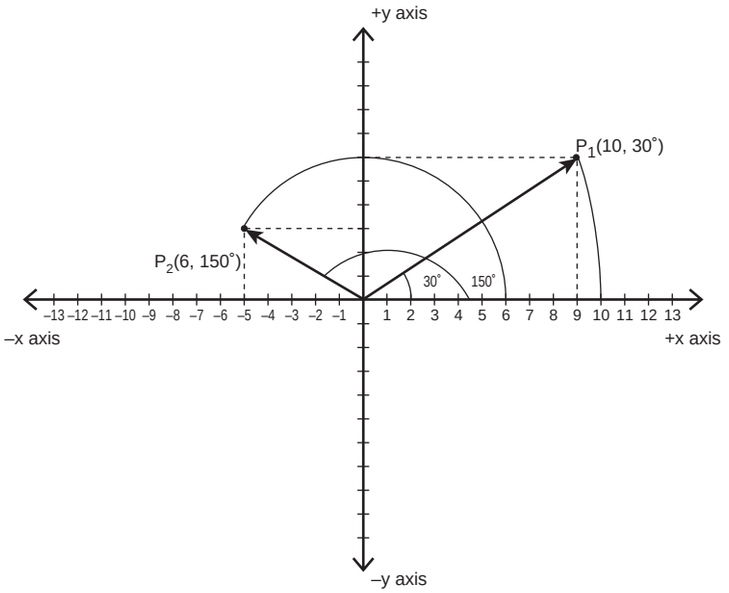


FIGURE 4.4 Locating points in a 2D polar system.

2D Polar Coordinates

The next type of coordinate system that two degrees of freedom supports is called *polar coordinates*. Polar coordinates are the entire basis of the game *Wolfenstein* and the technique of *ray-casting*. Polar coordinates are based on the desire to locate points in a 2D plane based on a heading and a distance, rather than an explicit (x,y) pair. Figure 4.3 depicts the standard 2D polar coordinate system. As you see, there are two variables used to locate a point: a distance r from the origin or *pole*, and a heading or angle θ (theta). Thus, the notation **p**(r, θ) would mean that the point **p** is located an angle θ degrees (or radians) relative to a reference ray (usually the x-axis) measured counter-clockwise and a distance r on that directional heading. Figure 4.4 shows examples of the points $\mathbf{p}_1(10,30°)$, and $\mathbf{p}_2(6,150°)$ using the x-y system as the reference.

Converting Between 2D Polar and Cartesian Coordinates

That's all fine and dandy, but polar coordinates are usually converted at some point into Cartesian coordinates to be useful. So how do we convert? Converting from Cartesian to polar and vice versa is easy if we just use a little geometry and the right triangle. Take a look at Figure 4.5—it shows a standard right triangle in quadrant I of a 2D Cartesian system. If we think of the tip of the hypotenuse as the "locator," or P, we can use the following formulas to convert from polar to Cartesian.

Playrix Ex. 1023, page 271

07 0672318350 CH04 05•02•2003 2:35 PM Page 247

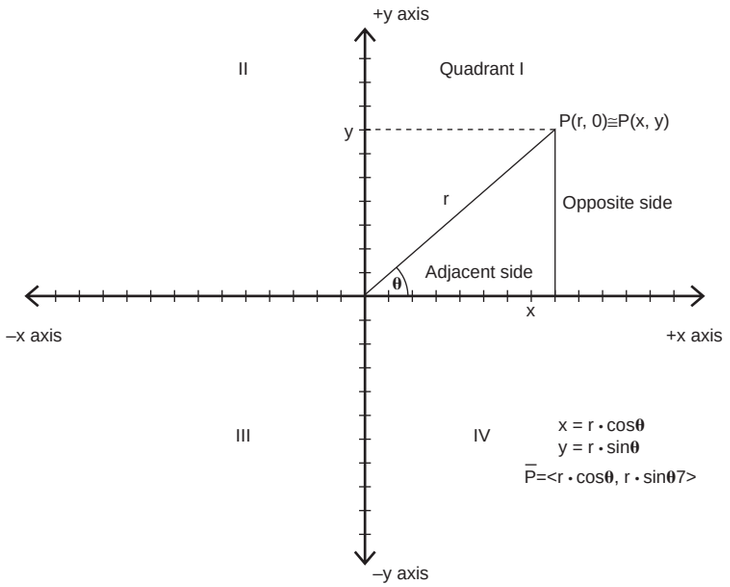


FIGURE 4.5 Geometrical interpretation of conversion between polar and Cartesian coordinates.

EQUATION 4.1 Converting 2D Polar Coordinates **p**(r,θ) to Cartesian Coordinates **p**(x,y)

```
x = r*cos(θ)
y = r*sin(θ)
```

TIP

If you are rusty on trigonometry, we will cover it a little later in the chapter, so for now just use the equations.

Converting from Cartesian to polar coordinates is a little more tricky. The problem is that we need the angle that the hypotenuse of the triangle formed by the point **p**(x,y) makes with the x-axis, along with its length. Again, trigonometry to the rescue. We can use the tangent to solve for the angle θ and the Pythagorean theorem to solve for *r*.

EQUATION 4.2 Converting 2D Cartesian Coordinates **p**(x,y) to Polar Coordinates **p**(r,θ)

Given: $x^2 + y^2 = r^2$

```
r = sqrt(x² + y²)
θ = tan⁻¹(y/x)
```

Playrix Ex. 1023, page 272

07 0672318350 CH04 05•02•2003 2:35 PM Page 248

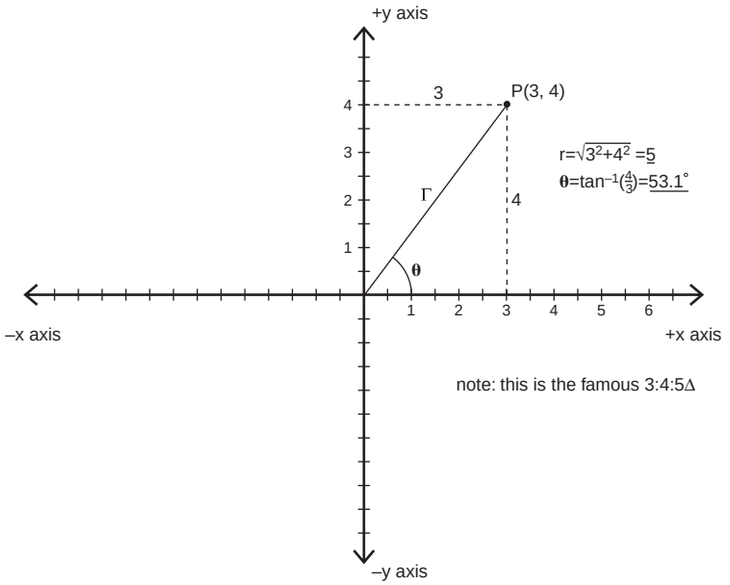


FIGURE 4.6 An example of converting from Cartesian to polar coordinates.

Let's give that a try for the point (3,4), which is located in QI and shown in Figure 4.6.

x = 3, y = 4, plugging into Equation 4.2:

```
r = sqrt(3² + 4²) = 5
θ = tan⁻¹(4/3) = 53.1°
```

Which, referring to Figure 4.6, looks about right.

TIP

If you are awake, you will notice that there is a *pole* (y goes to infinity) at x = 0! Or in other words, at the angles θ = 90° and θ = 270°, tangent is undefined (actually it's infinity [∞]). So to use Equation 4.2, you must test for x = 0 beforehand. In that case, you know the angle is 90°.

In conclusion, polar coordinates are very useful, and understanding them and how to convert back and forth is very important in many problems with trajectory, targeting, and navigation. If I had a dollar for every time I received an email about converting polar to Cartesian or vice versa, I would be a billionaire, so remember the equations—please! Otherwise, I'm going to start charging!

Playrix Ex. 1023, page 273

07 0672318350 CH04 05•02•2003 2:35 PM Page 249

3D Coordinate Systems

Now that you have a good grasp of 2D coordinates, let's move on to 3D coordinates. If you have been game programming for a while, you are probably already familiar with 3D systems. But I want to cover the cylindrical and spherical systems as well, because they will help us later on with texture mapping and other special effects. We will start off with the standard Cartesian 3D systems and work our way into cylindrical and spherical systems.

3D Cartesian Coordinates

The 3D Cartesian coordinate system is identical to the 2D system with the addition of the z-axis, for a total of three degrees of freedom based on three mutually *orthogonal* (perpendicular) axes. Therefore, to locate a point **p** in 3D-space, you need three coordinates: x, y, z, or in compact form, **p**(x,y,z). Additionally, the three axes form three planes: the x-y plane, the x-z plane, and the y-z plane, as shown in Figure 4.7. These planes are very important, and each divides space into two regions, or half spaces. This concept is important, and we will revisit it in a number of algorithms. Now for a little work—there are three planes; each plane divides space into two half spaces. How many unique sub-spaces are there in a 3-axis system? The answer is eight! Therefore, in a 3-space system, there are eight *octants*. This is shown in Figure 4.8.

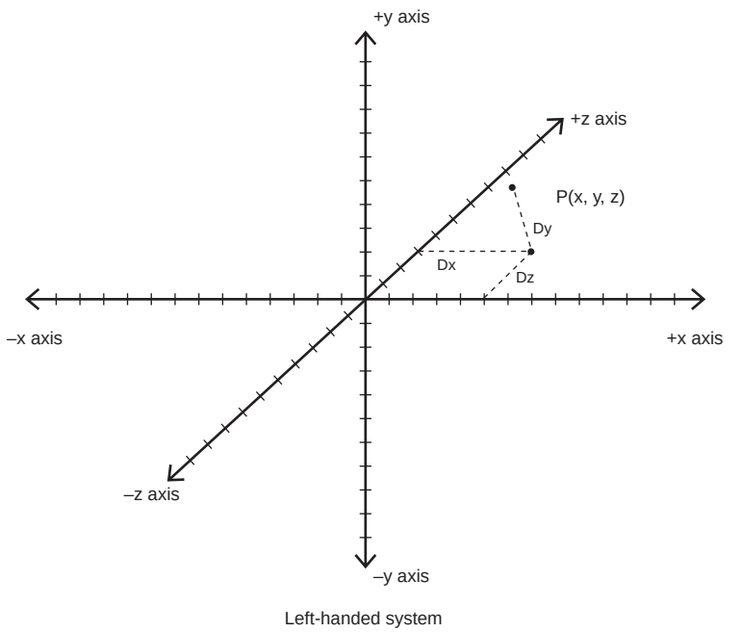


FIGURE 4.7 The 3D Cartesian coordinate system.

Playrix Ex. 1023, page 274

07 0672318350 CH04 05•02•2003 2:35 PM Page 250

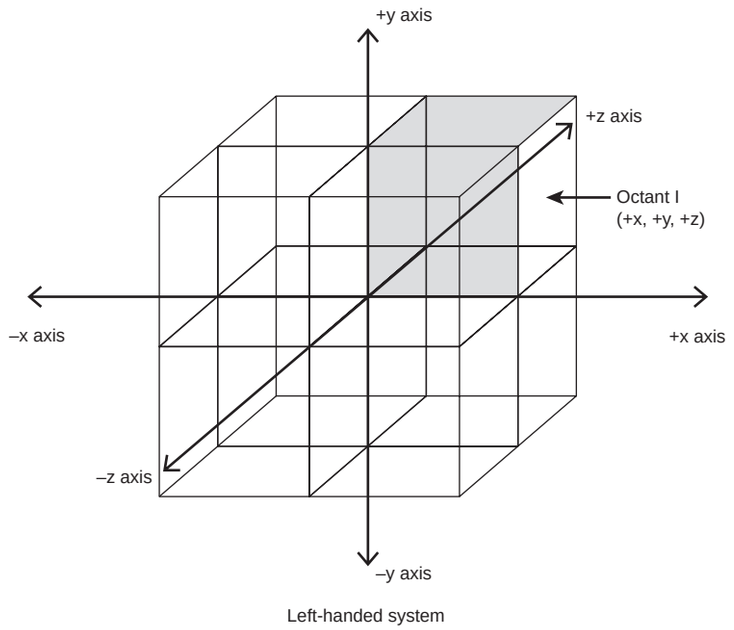


FIGURE 4.8 The octants of the 3D system.

This is all fantastic; however, this new z-axis creates a slight problem, as we must decide on two different orientations for the positive and negative half spaces for z, thus there are two distinct 3D Cartesian coordinate systems: the *left-handed system* and the *right-handed system*.

TIP

The names left-handed and right-handed come from the fact that if you wrap your right or left hand around the z-axis from x to y then your thumb will point in the direction of positive z.

The Left-Handed System

The left-handed system (LHS) is shown in Figure 4.9. In the LHS, the positive z-axis runs into the paper if we look straight down on it, or into the screen if we map x-y to the horizontal and vertical axes of the screen.

The Right-Handed System

The right-handed system (RHS) is shown in Figure 4.10. In the RHS, the positive z-axis runs out of the paper if we look straight down on it, and the negative z-axis runs into the paper. Thus, if we map the x-y to the screen as before, the negative z-axis runs into the screen and the positive z-axis out of the screen.

Playrix Ex. 1023, page 275

07 0672318350 CH04 05•02•2003 2:35 PM Page 251

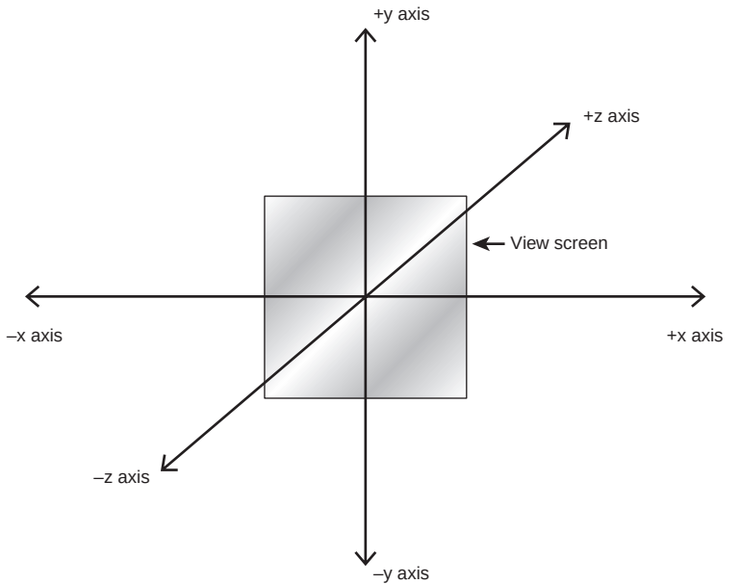


FIGURE 4.9 The left-handed 3D system.

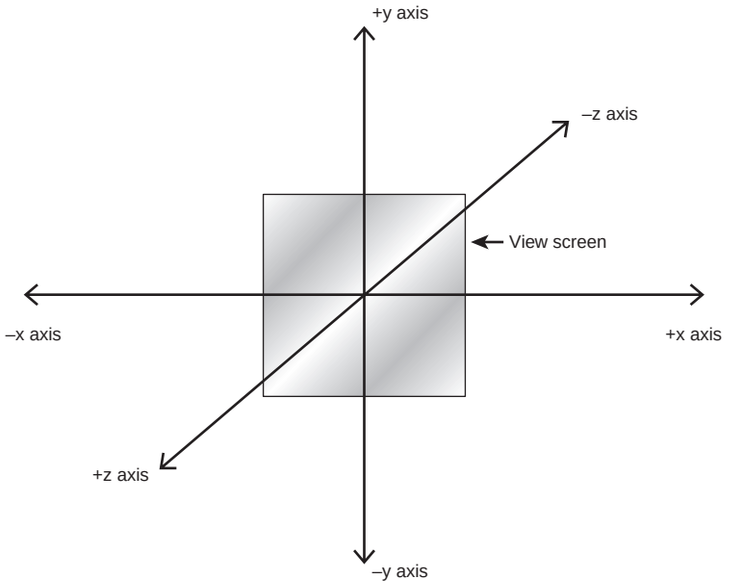


FIGURE 4.10 The right-handed 3D system.

Playrix Ex. 1023, page 276

07 0672318350 CH04 05•02•2003 2:35 PM Page 252

NOTE

There isn't any real difference between either system, but in this book we will primarily use the LHS for the 3D engine. However, I might use the RHS to make 3D figures easier to understand.

We will generally use 3D Cartesian coordinates for everything in this book, but sometimes it's easier to think in terms of angles or directions when solving a particular problem—working with 3D cameras, for example. Therefore, understanding the two other common 3D coordinate systems is a good idea, so let's take a look.

3D Cylindrical Coordinates

Interestingly enough, there are polar coordinates in 2D, but in 3D there are both cylindrical and spherical coordinates because we have one more degree of freedom. We are free to interpret the other two as we wish, giving us two more distinct coordinate systems. Of course, they are all mappable to each other, but each has its uses. The 3D *cylindrical coordinate system* is the closest to the 2D polar system, because it is basically nothing more than a 2D x-y polar system with z tacked onto it.

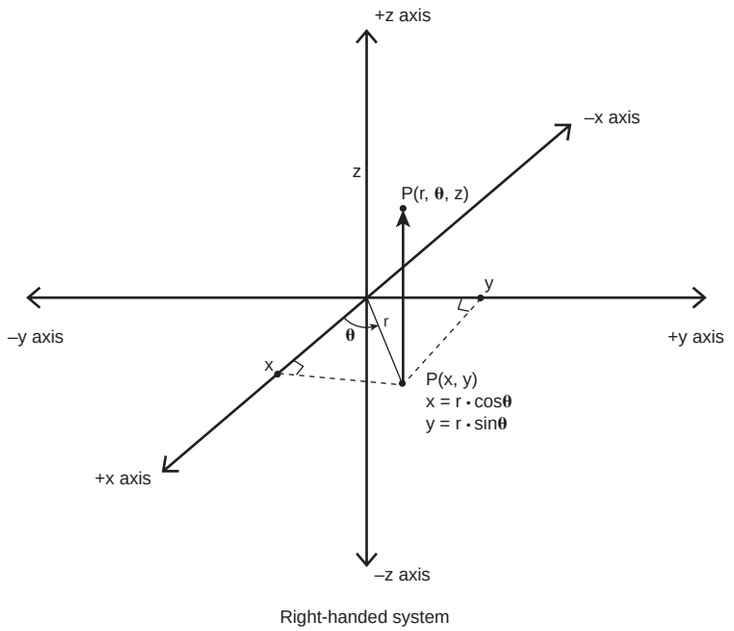


FIGURE 4.11 The cylindrical coordinate system.

Playrix Ex. 1023, page 277

07 0672318350 CH04 05•02•2003 2:35 PM Page 253

Take a look at Figure 4.11 to see the standard cylindrical coordinate system. You will notice that the x-y plane at z = 0 is the standard 2D polar system, and the cylindrical coordinate locates a point by first locating the 2D point **p**(r, θ) in the x-y axis at z = 0 and then "climbing" up the z-axis an amount z. Also, note that a right-handed system is used, and it's rotated on its side to make the illustration easier to see.

Converting Between 3D Cartesian and Polar Coordinates

Converting between 3D Cartesian coordinates and cylindrical coordinates is trivial. We just use the 2D transform and then let z = z.

EQUATION 4.3 Converting 3D Polar Coordinates **p**(r,θ,z) to Cartesian Coordinates **p**(x,y,z)

```
x = r*cos(θ)
y = r*sin(θ)
z = z
```


In converting Cartesian coordinates to cylindrical, we again first use the 2D Cartesian-to-polar transform and then once again let z = z.

EQUATION 4.4 Converting 3D Cartesian Coordinates **p**(x,y,z) to Polar Coordinates **p**(r,θ,z)

Given: $x^2 + y^2 = r^2$

```
r = sqrt(x² + y²)
θ = tan⁻¹(y/x)
z = z
```

CAUTION

Again, the problem for the tangent still exists at the angles θ = 90° and θ = 270°.

Cylindrical coordinates are handy for a number of problems, such as controlling the camera in a first-person shooter to environment mapping.

3D Spherical Coordinates

3D spherical coordinates are a little more complex than all the other coordinate systems. In general, a point is located via two angles and a distance from the origin of the system (usually x-y-z). Figure 4.12 shows a spherical coordinate system. To locate a point with spherical coordinates, you need two angles and the distance from the origin. Conventionally, this is written **p**(ρ,ϕ,θ); that is, "rho," "phi," and "theta," where:

Playrix Ex. 1023, page 278

07 0672318350 CH04 05•02•2003 2:35 PM Page 254

- ρ is the distance from the origin to the point **p**.
- ϕ is the angle that the directed line segment from the origin (**o**) to **p** makes with the positive z-axis (thus we will use a RHS system in this analysis).
- θ is the angle that the projection of the line from **o** to **p** makes on the x-y plane, which is just the standard 2D polar θ. Also, 0 <= θ <= 2π; that is, θ must be in the interval from 0 to 2π inclusive.

Hence, once again we see that we can derive spherical coordinates by using what we know about 2D polar coordinates. Let's see how to convert back and forth from 3D Cartesian and spherical coordinates, shall we?

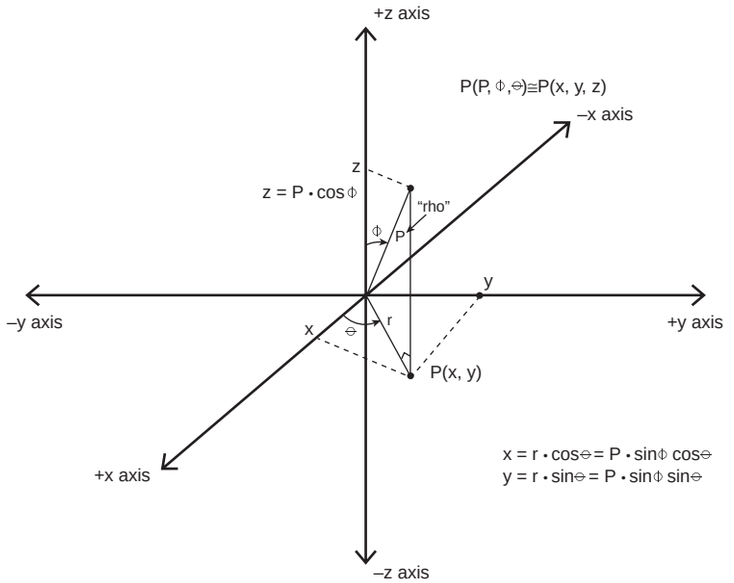


FIGURE 4.12 Spherical coordinates and conversion.

First, pay close attention to Figure 4.12, because the text description of the conversion is going to be confusing. To convert the spherical coordinates to 3D Cartesian coordinates (x,y,z), we can use a two-step process. First, we can project the line segment **o->p** onto the x-y plane and kill the z-axis. After we have done this, the problem becomes 2D and we already have a solution to it. Then we can back up and find z in terms of (ρ,ϕ,θ). This is the plan…

Playrix Ex. 1023, page 279

07 0672318350 CH04 05•02•2003 2:35 PM Page 255

EQUATION 4.5 Converting 3D Spherical Coordinates **p**(ρ,ϕ,θ) to Cartesian Coordinates **p**(x,y,z)

From the projection of the line segment **o->p** onto the x-y plane, we see that:

```
r = ρ*sin(ϕ)
z = ρ*cos(ϕ)
```

and in the x-y plane, we know that:

```
x = r*cos(θ)
y = r*sin(θ)
```

Therefore, substituting r into x,y and collecting equations gives:

```
x = ρ*sin(ϕ)*cos(θ)
y = ρ*sin(ϕ)*sin(θ)
z = ρ*cos(ϕ)
```


EQUATION 4.6 Converting 3D Cartesian Coordinates **p**(x,y,z) to Spherical Coordinates **p**(ρ,ϕ,θ)

Given: $x^2 + y^2 + z^2 = \rho^2$

Similarly: $x^2 + y^2 = r^2$

Therefore:

```
r = sqrt(x² + y²)
ρ = sqrt(x² + y² + z²)
θ = tan⁻¹(y/x)
```

And we can solve for ϕ after we have r and ρ from the relationship:

```
r = ρ*sin(ϕ)
```

Solving for ϕ gives:

```
ϕ = sin⁻¹(r/ρ)
```

Also, we can use:

```
z = ρ*cos(ϕ)
```

Therefore:

```
ϕ = cos⁻¹(z/ρ)
```

Playrix Ex. 1023, page 280

07 0672318350 CH04 05•02•2003 2:35 PM Page 256

Trigonometry

Trigonometry is the study of angles, shapes, and their relationships. Most trigonometry is based on the analysis of a *right triangle,* as shown in Figure 4.13. The right triangle has three interior angles and three sides. The angle opposite the 90 degree angle is usually used as the reference point, and is sometimes referred to as the *base angle*. From it, we can label the triangle. The side adjacent to the base angle is called the *adjacent side*. The side opposite the base angle is called the *opposite side*, and the side connecting them is called the *hypotenuse*. Don't underestimate the power of the triangle and similar triangles. I have seen few graphics algorithms that don't use triangles or similar triangles in some manner, so although it might seem trivial, expertise working with triangles is very important in graphics programming.

The Right Triangle

With all that in mind, let's just review some commonly known facts about triangles.

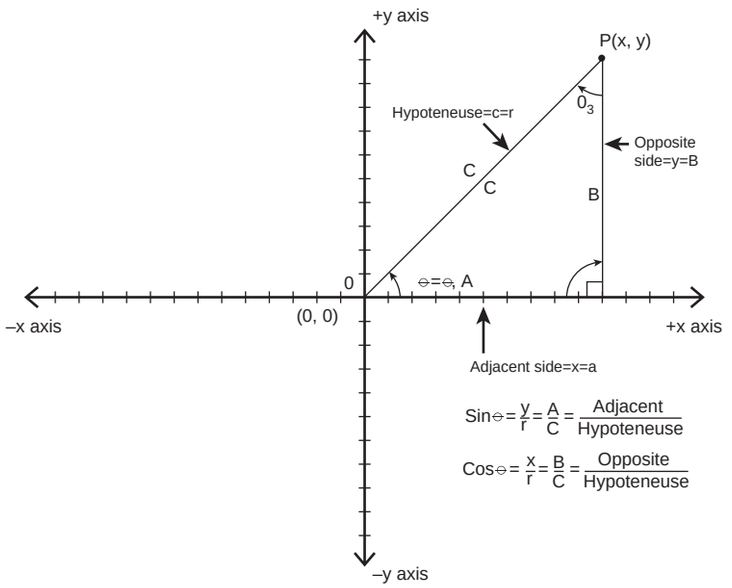


FIGURE 4.13 The right triangle.

NOTE

I will use *PI* and the symbol π interchangeably throughout the book. However, PI will usually mean the computer constant in the MATH.H header, whereas π means 3.14159….

Playrix Ex. 1023, page 281

07 0672318350 CH04 05•02•2003 2:35 PM Page 257

TABLE 4.4 Radians Versus Degrees

Angle in Degrees	Angle in Radians
360 degrees	2*PI radians (approx. 6.28 radians)
180 degrees	PI radians (approx. 3.14159 radians)
57.296 degrees	360 degrees/2*PI radians = 1 radian
1.0 degrees	2*PI radians/360 degrees = 0.0175 radians

Fact 1: There are 360 degrees in a complete circle, or *2*PI* radians. Hence, there are *PI* radians in 180 degrees. The computer functions *sin()* and *cos()* work in radians, ***NOT*** degrees—remember that! Table 4.4 lists the values.

Fact 2: The sum of the interior angles $\theta_1 + \theta_2 + \theta_3 = 180$ degrees, or *PI* radians.

Fact 3: Once again referring to the right triangle in Figure 4.13, the side opposite θ_1 is called the opposite side, the side below it is called the adjacent side, and the long side is called the hypotenuse.

Fact 4: The sum of the squares of the sides of a right triangle equals the square of the hypotenuse. This is called the *Pythagorean Theorem*. Mathematically, we write it like this:

$$\text{hypotenuse}^2 = \text{adjacent}^2 + \text{opposite}^2$$

or sometimes using a, b, and c for dummy variables:

$$c^2 = a^2 + b^2$$

Therefore, if you have two sides of a right triangle, you can always find the third.

Fact 5: There are three main trigonometric ratios that mathematicians like to use. They are called the *sine*, *cosine*, and *tangent*. They are defined as:

$$\cos(\theta) = \frac{\text{adjacent side}}{\text{hypotenuse}} = \frac{x}{r}$$

DOMAIN: 0 <= θ <= 2*PI

RANGE: -1 to 1

$$\sin(\theta) = \frac{\text{opposite side}}{\text{hypotenuse}} = \frac{y}{r}$$

DOMAIN: 0 <= θ <= 2*PI

RANGE: -1 to 1

Playrix Ex. 1023, page 282

07 0672318350 CH04 05•02•2003 2:35 PM Page 258


```
             sin(θ)        opposite/hypotenuse
tan(θ)  = ------------ = --------------------
             cos(θ)        adjacent/hypotenuse

       opposite         y
     = ------------ = ----- = slope = M
       adjacent         x
```

DOMAIN: -PI/2 <= θ <= PI/2

RANGE: -infinity to +infinity

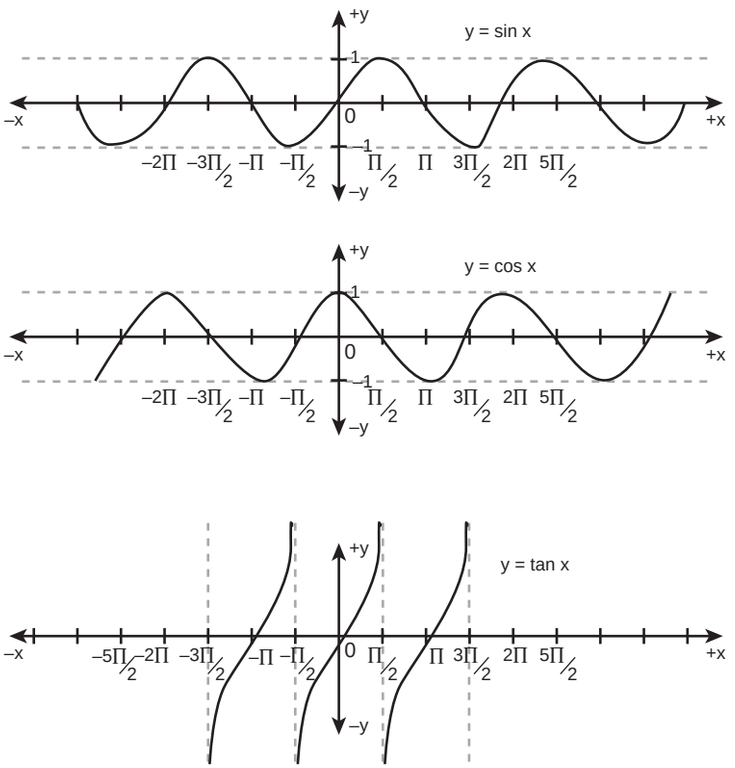


FIGURE 4.14 Graphs of basic trigonometric functions.

Playrix Ex. 1023, page 283

07 0672318350 CH04 05•02•2003 2:35 PM Page 259

Figure 4.14 shows graphs of all the functions. Notice that all the functions are periodic (repeating) and that *sin(θ)* and *cos(θ)* have periodicity of *2*PI* while tangent has periodicity of *PI*. Also, notice that *tan(θ)* goes to + or -infinity whenever θ modulus *PI* is *PI/2*.

MATH

You might note the use of the terms *DOMAIN* and *RANGE*. These simply mean the input and the output, respectively.

Inverse Trigonometric Functions

Many times when using trig functions like sine, cosine, and tangent, you create a formula based on some geometry and need the initial angle back. That is, you want to use the function in reverse to find the angle that some given ratio represents in terms of sine, cosine, or tangent. This is called the inverse, and is written using a superscript -1:

$\theta = \cos^{-1}(x)$, where $0 <= \theta <= 2*PI$ and $-1 <= x <= 1$

$\theta = \sin^{-1}(x)$, where $0 <= \theta <= 2*PI$ and $-1 <= x <= 1$

$\theta = \tan^{-1}(x)$, where $-PI/2 <= \theta <= PI/2$ and $-\infty < x < +\infty$

NOTE

Of course, on a computer there usually isn't a representation for infinity, but in this context we mean numbers sufficiently large that for all intents and purposes they are many times larger than any other number in the number space we are considering. For example, if your game is working with a grid of 1000x1000, then 100,000,000 is close enough to infinity.

Using the preceding inverse functions is easy: Just plug in a number for x and solve for θ (actually, let the computer do it). However, there is a catch. Although standard sine, cosine, and tangent are slow (even on a computer), the inverse functions are 10 times slower! So, make sure you really need to know the actual angle. Many times you can just use the x itself in your algorithms. Of course, you can always use tricks like look-up tables and interpolation if you really need angles back.

Trigonometric Identities

Now, there are about a gazillion trigonometric identities and tricks, and it would take a math book to prove them all, so I'm just going to give you a table of the ones that a game programmer should know. Table 4.5 lists some other trigonometric ratios, as well as some neat identities.

Playrix Ex. 1023, page 284

07 0672318350 CH04 05•02•2003 2:35 PM Page 260

TABLE 4.5 Useful Trigonometric Identities

Reciprocal Functions
Cosecant: $\csc(\theta) = 1/\sin(\theta)$
Secant: $\sec(\theta) = 1/\cos(\theta)$
Cotangent: $\cot(\theta) = 1/\tan(\theta)$
Pythagorean Theorem in terms of trig functions:
$\sin(\theta)^2 + \cos(\theta)^2 = 1$
Conversion identity:
$\sin(\theta) = \cos(\theta - PI/2)$
Reflection identities:
$\sin(-\theta) = -\sin(\theta)$
$\cos(-\theta) = \cos(\theta)$
Angle Addition identities:
$\sin(\theta_1 + \theta_2) = \sin(\theta_1)*\cos(\theta_2) + \cos(\theta_1)*\sin(\theta_2)$
$\cos(\theta_1 + \theta_2) = \cos(\theta_1)*\cos(\theta_2) - \sin(\theta_1)*\sin(\theta_2)$
$\sin(\theta_1 - \theta_2) = \sin(\theta_1)*\cos(\theta_2) - \cos(\theta_1)*\sin(\theta_2)$
$\cos(\theta_1 - \theta_2) = \cos(\theta_1)*\cos(\theta_2) + \sin(\theta_1)*\sin(\theta_2)$

Of course, you could derive identities until you turn many shades of green. In general, identities help you reduce complex trigonometric formulas into simpler ones, so you don't have to do the math. Hence, when you come up with an algorithm based on sine, cosine, tangent, and so on, always take a look in a trigonometry book and see whether you can simplify your math, so that fewer computations are needed to get to the result. Remember: speed, speed, speed!!!

TRICK

When we get to matrix transformations, we might decide to concatenate a number of operations together, like translation, rotation, and scaling. The rotation operation itself might be a concatenation of two or more rotation transformations, all in terms of sine and cosine. Hence, if we look at the results of the matrix multiplications, we can simplify the matrix using trigonometric identities and write a specific function that performs the transformation, rather than using general matrix multiplications blindly.

Playrix Ex. 1023, page 285

07 0672318350 CH04 05•02•2003 2:35 PM Page 261

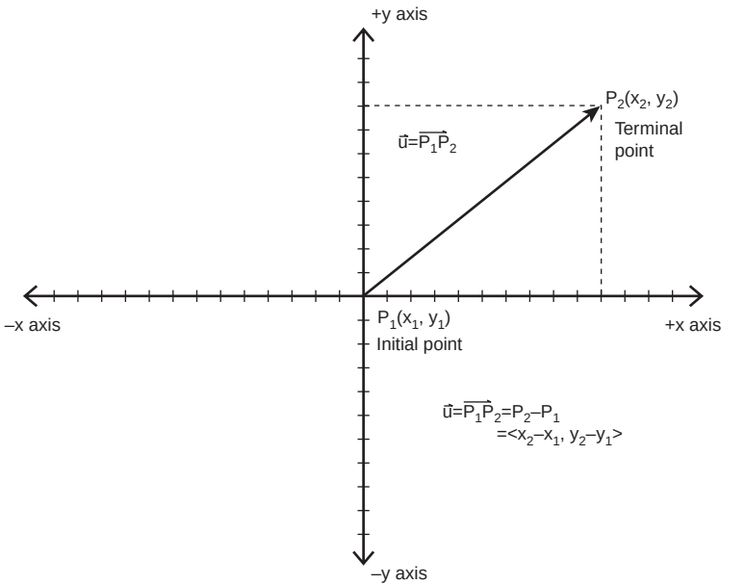


FIGURE 4.15 Definition of a vector.

Vectors

Vectors are the basis for all 3D algorithms, and thus are very important to a game programmer. Vectors are basically nothing more than a collection of two or more individual components that represent directed line segments in the 2- and 3-component case (which we are interested in). They are defined by an *initial point* and a *terminal point*, as shown in Figure 4.15. Referring to Figure 4.15, we see a vector **u** defined by the two points **p1** (the initial point) and **p2** (the terminal point). The vector **u** = $<u_x,u_y>$ is from **p1**(x1,y1) to **p2**(x2,y2). To compute **u**, you simply subtract the initial point from the terminal point:

```
U = p2 - p1 = (x2-x1, y2-y1) = <ux, uy>
```

Another way to represent the vector from **p1** to **p2** is to use bar notation by placing the two endpoints adjacent to each other and placing a bar over them, like this:

$\overline{p_1p_2}$

which means the vector formed from point **p1** to **p2**. Also, we usually represent vectors by bolded lowercase (or uppercase) letters, like **u**, and the components are written within angled brackets, like $<u_x,u_y>$.

Playrix Ex. 1023, page 286

07 0672318350 CH04 05•02•2003 2:35 PM Page 262

Okay, so a vector represents a directed line segment from one point to another, but that segment can represent a lot of concepts, such as *velocity*, *acceleration*, or whatever. However, be warned: Once defined, vectors are always relative to the origin. This means that after you create a vector from point **p1** to **p2**, the initial point in vector space is always at (0,0), or (0,0,0) in 3D. This doesn't matter because the math takes care of everything, but if you think about it, it makes sense. A vector consists of two numbers in 2D and three numbers in 3D space, so it really only defines an endpoint in 2D or 3D space. Hence, the starting point is always thought of as the origin. This doesn't mean that you can't translate vectors around and perform various geometrical operations with the vectors themselves; it just means that you need to keep in mind what a vector really is.

The cool things about vectors are the operations that we can perform on them. Because vectors are really sets of ordered numbers, we can perform many of the standard mathematical operations on them by performing the mathematical operation on each component independently.

NOTE

Vectors can have any number of components. Usually, in computer graphics we deal with 2D and 3D vectors; that is, vectors of the form **a** = <x,y>, **b** = <x,y,z>. An *n*-dimensional vector has the form

c = <c1, c2, c3, …, cn>

n dimensional vectors are used to represent sets of variables rather than geometrical space, because after 3D we enter *Hyperspace*, as normal space only has three spatial dimensions. Right, Mr. Mallory?

Vector Length

The first thing that will pop up time and time again when working with vectors is how to compute the length of a vector or its magnitude. The length of a vector is called the *norm* and is represented by two vertical bars on either side of the vector (at least in this book), such as $|\mathbf{u}|$ and read as, "the length of **u**." The length is computed as the distance from the origin to the tip of the vector, or in general as the square root of the sum of the squares of the components. Hence, we can use the standard Pythagorean Theorem to find the length in the 2D and 3D case. Hence, $|\mathbf{u}|$ is equal to the following:

EQUATION 4.7 The Length of a 2D Vector $\mathbf{u} = \langle u_x, u_y \rangle$

$$|\mathbf{u}| = \text{sqrt}(u_x^2 + u_y^2)$$

And if **u** happened to be a 3D vector, the length would be:

EQUATION 4.8 The Length of a 3D Vector $\mathbf{u} = \langle u_x, u_y, u_z \rangle$

$$|\mathbf{u}| = \text{sqrt}(u_x^2 + u_y^2 + u_z^2)$$

Playrix Ex. 1023, page 287

07 0672318350 CH04 05•02•2003 2:35 PM Page 263

Normalization

After you have the length of a vector, you can do something interesting—*normalize* it, or in other words, shrink the vector to make sure that its length is 1.0 while maintaining the direction. Unit vectors have a lot of nice properties, just like the scalar 1.0. For example, sometimes we might perform a number of operations and the resulting vector is of interest to us, but we don't care about its length; we just need a vector of length 1.0 in the direction of the computed vector. Normalization can give us this. Given a vector $\mathbf{n} = \langle n_x, n_y \rangle$, the normalized version of $\mathbf{n}$ is usually written as $\mathbf{n}$ with a "hat" over it, something like this:

$$\hat{\mathbf{n}}$$

But, again that's pressing my luck with the editors in layout, so we will use lowercase italic (along with a superscript prime symbol sometimes for unit vectors) like this:

$$n \text{ or } n'$$

In any case, the normalized version of a vector $\mathbf{n}$ is computed like this:

EQUATION 4.9 Normalizing a Vector

$$n' = \mathbf{n} / |\mathbf{n}|$$

Very simple. The normalized version of a vector is simply the vector divided (multiplied by the inverse) by the length of the vector.

Scalar Multiplication

The first operation that you might want to perform on a vector is scaling. For example, suppose you have a vector representing speed and you want to go faster (or slower). Scaling is performed by multiplying each component by a single scalar number. For example:

Let $\mathbf{u} = \langle u_x, u_y \rangle$ and k be a real constant

$$k*\mathbf{u} = k*\langle u_x, u_y \rangle = \langle k*u_x, k*u_y \rangle$$

Figure 14.16 shows the scaling operation graphically. In addition, if you want to invert the direction of a vector, you can multiply any vector by –1 and invert the vector as shown in Figure 4.17, or mathematically:

Let

$$\mathbf{u} = \langle u_x, u_y \rangle$$

then the vector in the opposite direction of $\mathbf{u}$ is

$$-1*\mathbf{u} = -1*\langle u_x, u_y \rangle = \langle -u_x, -u_y \rangle$$

Playrix Ex. 1023, page 288

07 0672318350 CH04 05•02•2003 2:35 PM Page 264

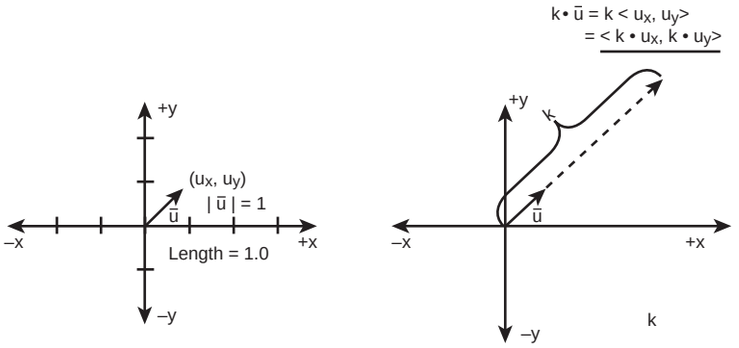


FIGURE 4.16 Vector scaling.

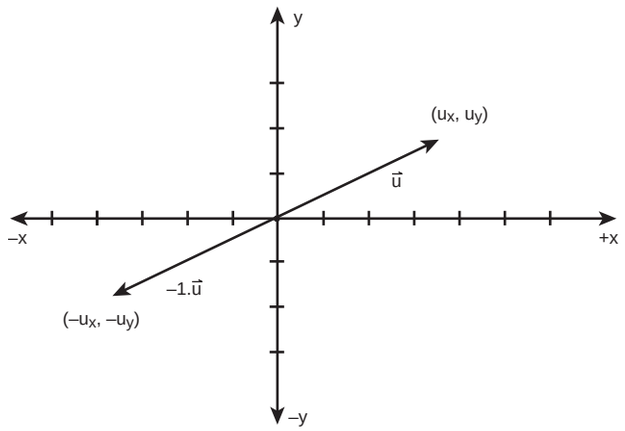


FIGURE 4.17 Vector inversion.

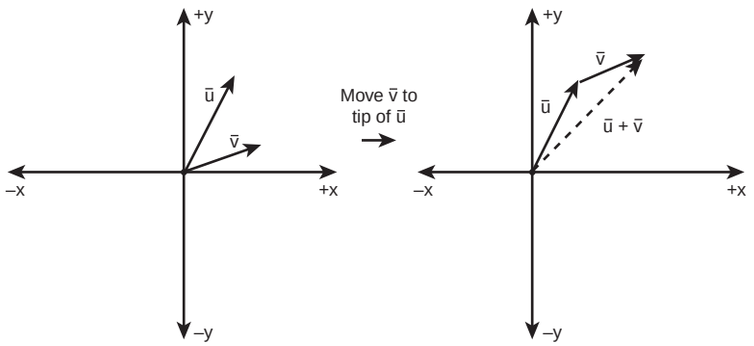


FIGURE 4.18 Vector addition.

Playrix Ex. 1023, page 289

07 0672318350 CH04 05•02•2003 2:35 PM Page 265

Vector Addition

To add two (or more) vectors together, all you have to do is add the respective components together. Figure 4.18 illustrates this graphically for the 2D case (3D would tax my artistic abilities). Vector **u** is added to **v**, and the result is **r**. Notice how the addition was performed geometrically: I took **v** and moved it to the terminal point of **u** and then drew the other side of the triangle. Geometrically, this operation is equivalent to the following mathematical operation:

$$\mathbf{u} + \mathbf{v} = \langle u_x, u_y \rangle + \langle v_x, v_y \rangle = \langle u_x+v_x, u_y+v_y \rangle$$

Thus, to add any number of vectors together on graph paper, you can simply add them "tip to tail," and when you add them all up, the vector from the origin to the last tip is the result.

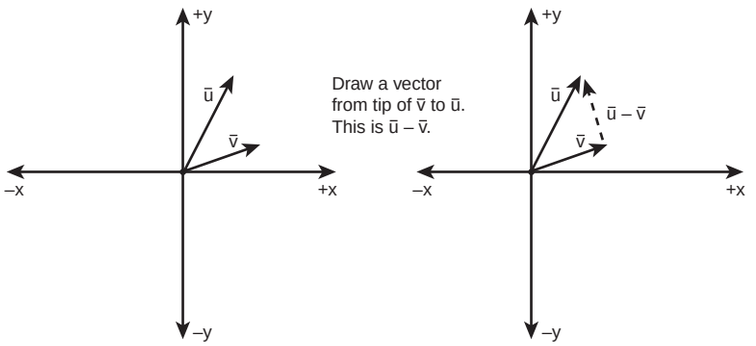


FIGURE 4.19 Vector subtraction.

Vector Subtraction

Vector subtraction is really vector addition with the opposite pointing vector; however, it is sometimes illustrative to see subtraction graphically. Take a look at Figure 4.19 to see **u** – **v** and **v** – **u**. Notice that **u** – **v** is computed by drawing a vector from **v** to **u**, and **v** – **u** is computed by drawing a vector from **u** (as the initial point) to **v** (the terminal point). Mathematically, we have:

$$\mathbf{u} - \mathbf{v} = \langle u_x, u_y \rangle - \langle v_x, v_y \rangle = \langle u_x-v_x, u_y-v_y \rangle$$

which might be easier to remember, but a piece of graph paper can sometimes be a much better "computer" than doing the math manually because you can visualize the data quicker. Hence, it's a good idea to know how to add and subtract vectors on graph paper when you are working out rendering algorithms—trust me!

Playrix Ex. 1023, page 290

07 0672318350 CH04 05•02•2003 2:35 PM Page 266

The Inner Product, or the "Dot" Product

At this point you might be asking, "Can you multiply two vectors together?" The answer is yes, but the straight component-wise multiplication represented by the symbol "⊗ " isn't very useful, it turns out:

$$\mathbf{u} \otimes \mathbf{v} = \langle u_x * v_x,\ u_y * v_y \rangle$$

However, another kind of multiplication known as the dot product is very useful. It's defined like this:

EQUATION 4.10 The Dot Product

$$\mathbf{u} \cdot \mathbf{v} = u_x * v_x + u_y * v_y$$

The dot product, usually represented by a dot (.), is computed by summing the products of the individual terms into a single scalar, rather than just multiplying the components and keeping the result in vector form. You might ask, what does the dot product do for us? We don't even have vectors anymore! True, but the dot product is also equal to this expression:

EQUATION 4.11 The Dot Product Related to the Angle Between the Vectors

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| * |\mathbf{v}| * \cos(\theta)$$

This states that **u** dot **v** is equal to the length of **u** multiplied by the length of **v** multiplied by the cosine of the angle between the vectors. If we combine the two different expressions, we get this:

$$\mathbf{u} \cdot \mathbf{v} = u_x * v_x + u_y * v_y$$
$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| * |\mathbf{v}| * \cos(\theta)$$
$$u_x * v_x + u_y * v_y = |\mathbf{u}| * |\mathbf{v}| * \cos\ \theta$$

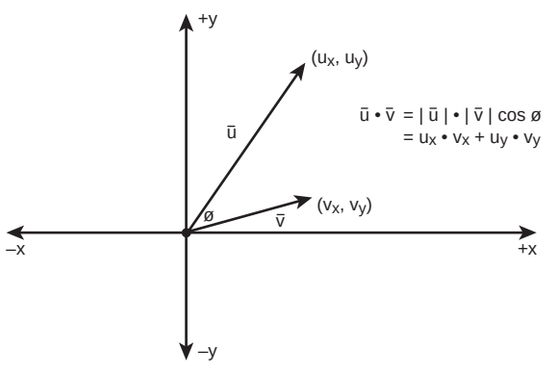


FIGURE 4.20 The dot product.

Playrix Ex. 1023, page 291

07 0672318350 CH04 05•02•2003 2:35 PM Page 267

This is a very interesting formula—it basically gives us a way to compute the angle between two vectors, as shown in Figure 4.20, and that's a really useful operation. If you can't see that, then take a look at the equation after rearranging and taking the inverse cosine of both sides:

EQUATION 4.12 Computing the Angle Between Two Vectors

$$\theta = \cos^{-1} (u_x * v_x + u_y * v_y / |\mathbf{u}| * |\mathbf{v}|)$$

or more compactly, we assume that (**u**.**v**) means ($u_x * v_x + u_y * v_y$) and just write

$$\theta = \cos^{-1} (\mathbf{u}.\mathbf{v} / |\mathbf{u}| * |\mathbf{v}|)$$

This is a very powerful tool and the basis of many 3D graphics algorithms. The cool thing is that if the **u** and **v** are already unit vectors—that is, the length of **u** and **v** are already 1.0—their product is 1.0 and the formula simplifies even more to

$$\theta = \cos^{-1} (\mathbf{u}.\mathbf{v}), \text{ for } |\mathbf{u}|=|\mathbf{v}| = 1.0$$

And here are a couple interesting facts about the dot product:

Fact 1: If the angle between **u** and **v** is 90 (perpendicular), then **u**.**v** = 0.

Fact 2: If the angle between **u** and **v** is < 90 (acute), then **u**.**v** > 0.

Fact 3: If the angle between **u** and **v** is > 90 (obtuse), then **u**.**v** < 0.

Fact 4: If **u** and **v** are equal, then $\mathbf{u}.\mathbf{v} = |\mathbf{u}|^2 = |\mathbf{v}|^2$.

These facts are all shown graphically in Figure 4.21.

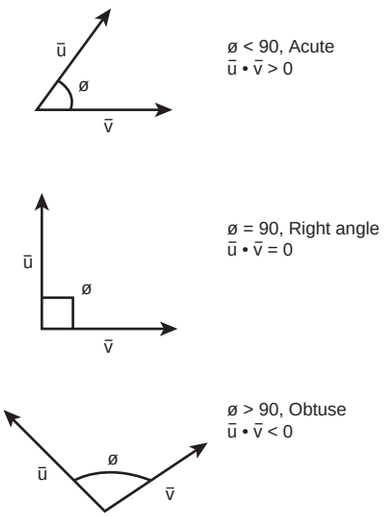


FIGURE 4.21 Angles and their relationship to the dot product.

Playrix Ex. 1023, page 292

07 0672318350 CH04 05•02•2003 2:35 PM Page 268

Additionally, the dot product can be used for many other computations. One that is of great use in computer graphics and game programming is computing the components of one vector (or its projection) along a given test vector's direction. Figure 4.22 shows this graphically. For example, suppose you have a vector **v** that represents the trajectory of one character in your game, and you have a vector **u** that represents the trajectory of another character in your game. There are many cases when you want to know how much **u** is pointing in the direction of **v**. In other words, you want the vector projection of **u** onto **v**, which is called $\mathbf{Proj}_{\mathbf{v}}\,\mathbf{u}$. We can compute this vector using the dot product.

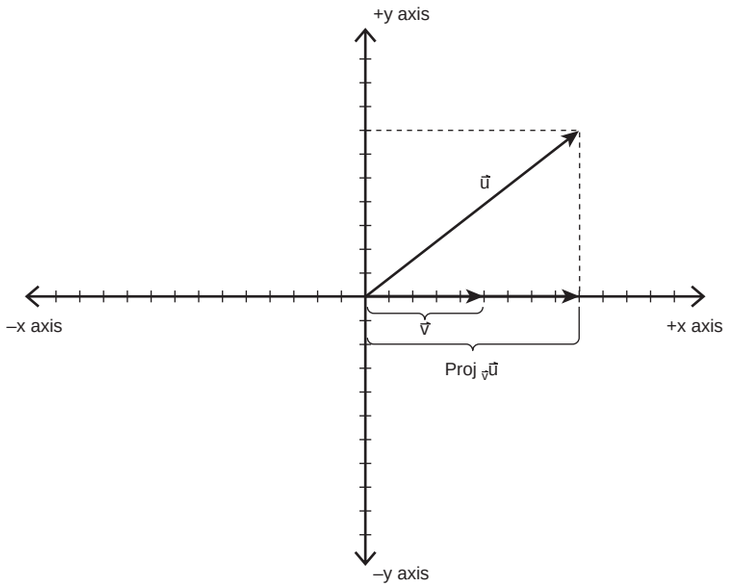


FIGURE 4.22 The projection of **u** on **v**.

Referring to Figure 4.22, we have both **u** and **v** and we want the length of the parallel projection of **u** onto **v** (we could just as well ask for the perpendicular length), then we can multiply this by the unit vector in the direction of **v** if we need a vector quantity. To compute this, we can use the dot product **u**.**v**. However, **v** must have length 1.0; that is, be normalized. Hence, the final equation for the vector of the projection **u** onto **v** is:

EQUATION 4.13 The Length of the Parallel Projection of **u** onto **v**

$$\mathbf{Proj}_{\mathbf{v}}\,\mathbf{u} = \frac{(\mathbf{u} \cdot \mathbf{v}) * \mathbf{v}}{|\mathbf{v}| * |\mathbf{v}|}$$

Playrix Ex. 1023, page 293

07 0672318350 CH04 05•02•2003 2:35 PM Page 269

Let's see if we can understand what's happening here. In the numerator, we have **u**.**v**, which we know means

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| * |\mathbf{v}| * \cos(\theta)$$
$$u_x * v_x + u_y * v_y = |\mathbf{u}| * |\mathbf{v}| * \cos\theta$$

Or in other words, **u**.**v** is equivalent to the length of **u** multiplied by the length of **v** times the cosine of the angle between them. In the context of this problem, we know that we are looking for the vector projection of **u** on **v**, but we need to convert **v** into a unit vector—that's what one of the |**v**| terms in the denominator does. You'll notice that I have deliberately aligned the **v**/|**v**| terms so you can see them. Thus, the |**v**| term is 1.0 and the dot product can be rewritten like this:

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| * 1 * \cos(\theta) = |\mathbf{u}| * \cos(\theta)$$

This is key. It states that as the angle θ becomes 0° (that is, as the vector **u** becomes co-linear with **v**), then |**u**|*cos(θ) becomes |**u**|, which is what we want. As the vectors align in the direction of **v**, then we should expect to get the total length of **u**, and we do. Now one last thing is up in the air—we need a vector, not just a magnitude! But this is easy. All we do is multiply the magnitude term

$$\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{v}|}$$

by the unit vector in the direction of **v**, and we are done. And the final result is exactly what we started with:

$$\text{Proj}_{\mathbf{v}}\,\mathbf{u} = \frac{(\mathbf{u} \cdot \mathbf{v}) * \mathbf{v}}{|\mathbf{v}| * |\mathbf{v}|}$$

The point of this exercise is to show how the dot product works to solve for projections. Basically, any time you want to find the amount that one vector is pointing in another vector's direction, you can use the dot product. However, remember that the test vector must be converted to a unit vector; **v** in this case.

Multiplicative Laws for Dot Products

Under normal multiplication of scalars, the associative, distributive, and commutative laws all work great. On the other hand, in vector space, sometimes these laws don't all work as you might think. Here's a list that encompasses all the laws relating to the dot product, so you don't do anything weird with them:

Playrix Ex. 1023, page 294

07 0672318350 CH04 05•02•2003 2:35 PM Page 270

Given that **u**, **v**, and **w** are vectors and k is a scalar:

(a) **u** . **v** = **v** . **u**

(b) **u** . (**v** + **w**) = (**u** . **v** + **u** . **w**)

(c) k*(**u** . **v**) = (k***u**) . **v** = (**u** . (k***v**))

Note: the dot product always has higher precedence than addition.

The Cross Product

The next type of multiplication that can be applied to vectors is called the *cross product*. However, the cross product only makes sense on vectors with three or more components. Therefore, in our case we will use 3D space vectors as an example. Given $\mathbf{u} = \langle u_x, u_y, u_z \rangle$ and $\mathbf{v} = \langle v_x, v_y, v_z \rangle$, the cross product written **(u × v)** is defined as:

EQUATION 4.14 The Cross Product in Terms of the Normal Vector and the Angle Between **u** and **v**

```
u x v = ¦u¦*¦v¦*sin(θ) * n
```

Alrighty then! Let's take this apart piece by piece. |**u**| denotes the length of **u**, |**v**| denotes the length of **v**, and sin(θ) is the sine of the angle between the vectors. Thus the product (|**u**|*|**v**|*sin(θ)) is a scalar; that is, a number. Then we multiply it by ***n***. But, what is ***n***? ***n*** is a unit normal vector, meaning that it's perpendicular to both **u** and **v** and has length 1.0. Figure 4.23 shows this graphically.

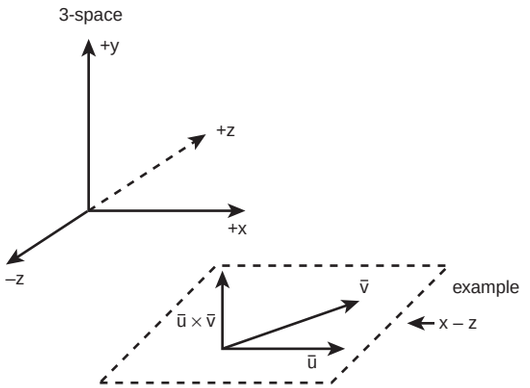


FIGURE 4.23 The cross product depicted graphically.

Playrix Ex. 1023, page 295

07 0672318350 CH04 05•02•2003 2:35 PM Page 271

So the cross product tells us something about the angle between **u** and **v**, and the normal vector to both **u** and **v**. But without another equation, we aren't going to get anywhere. The question is, how to compute the normal vector from **u** and **v**? The answer is with another definition of the cross product.

The cross product is also defined as a very special vector product. However, it's hard to show without matrices, so bear with me. Assume that you want to compute the cross product of **u** and **v** (**u** × **v**): You would build a matrix like this:

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

where **i**, **j**, and **k** are unit vectors parallel to the x-, y-, and z-axes, respectively.

Then to compute the cross product of **u** and **v**, you perform this multiplication:

EQUATION 4.15 Computing the Normal Vector Between Two Vectors **u** and **v**

$$\mathbf{n} = (u_y*v_z - v_y*u_z)*\mathbf{i} - (u_x*v_z - v_x*u_z)*\mathbf{j} + (u_x*v_y - v_x*u_y)*\mathbf{k}$$

n is just a linear combination of three scalars, each multiplied by mutually orthogonal (perpendicular) unit vectors that are each parallel to the x-, y-, and z-axes respectively. Thus we can forget **i**, **j**, and **k** and rewrite the equation as

$$\mathbf{n} = \langle u_y*v_z - v_y*u_z,\ -u_x*v_z + v_x*u_z,\ u_x*v_y - v_x*u_y \rangle$$

n is the normal vector to both **u** and **v**; however, it is not necessarily a unit vector (if **u** and **v** were both unit vectors, **n** would be), so we must normalize it to find **n**. After this is done, we can plug everything into our cross product equation and do what we will.

In practice however, few people ever use the **u** x **v** = |**u**|*|**v**|*sin(θ)***n** formula; they simply use the matrix form to find the normal vector because θ is usually unknown. Once again, we see normalized vectors are very important in 3D graphics, and you will use them for lighting, back-face culling, defining planes, to comparing the orientation of polygons, and for collision detection, rendering, and so forth.

Multiplicative Laws for Cross Products

Here's a list that encompasses all the laws relating to the cross product, so you don't do anything weird with them:

Given that **u**, **v**, and **w** are vectors and k is a scalar:

(a) **u** × **v** = -(**v** × **u**) (very important!)

(b) **u** × (**v** + **w**) = (**u** × **v**) + (**u** × **w**)

(c) (**u** + **v**) × **w** = (**u** × **w**) + (**v** × **w**)

(d) k*(**u** × **v**) = (k***u**) × **v** = **u** × (k***v**)

Playrix Ex. 1023, page 296

07 0672318350 CH04 05•02•2003 2:35 PM Page 272

The Zero Vector

Although you probably won't use the zero vector much, it does exist. The zero vector has zero length, no direction, and is simply a point, if you want to get technical. Thus in 2D, the zero vector is <0,0>, and in 3D it's <0,0,0>, and so on for higher dimensions.

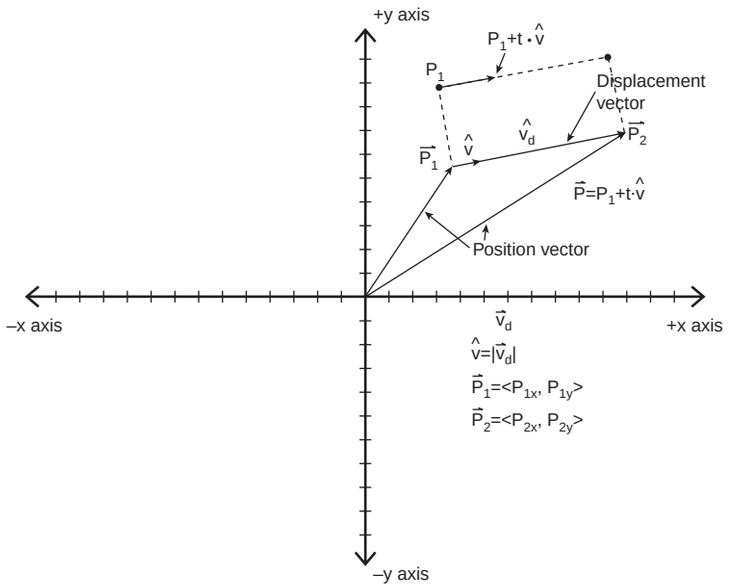


FIGURE 4.24 Position and displacement vectors.

Position and Displacement Vectors

The next topic I want to talk about is *position* and *displacement vectors*. These are really useful when tracing out geometrical entities like lines, segments, curves, and so on. Take a look at Figure 4.24—it depicts a position vector that can be used to represent a line segment. The line segment is from **p1** to **p2**, $\mathbf{v_d}$ is the displacement vector from **p1** to **p2**, and ***v*** is a unit vector in the direction of **p1** to **p2**. We then construct the vector **p** to trace out the segment. **p** looks like this mathematically:

```
p = p1 + t*v
```

where t is a parameter that varies from 0 to $|\mathbf{v_d}|$. If $t = 0$, then we have

```
p = p1 + 0*v = <p1> = <p1x, p1y>
```

Playrix Ex. 1023, page 297

07 0672318350 CH04 05•02•2003 2:35 PM Page 273

Thus at $t = 0$, **p** is pointing to the beginning of the segment. On the other hand, when $t = |\mathbf{v_d}|$ we have

$$\mathbf{p} = \mathbf{p1} + |\mathbf{v_d}|*\mathbf{v} = \mathbf{p1} + \mathbf{v_d} = <\mathbf{p1}+ \mathbf{v_d}>$$
$$= <p1_x+v_x,\ p1_y+v_y>$$
$$= \mathbf{p2} = <p2_x,\ p2_y>$$

Vectors as Linear Combinations

As you saw in the cross product, calculation vectors can also be written in the notation

$$\mathbf{U} = u_x*\mathbf{i} + u_y*\mathbf{j} + u_z*\mathbf{k}$$

where **i**, **j**, and **k** are unit vectors parallel to the x-y-z axes. There is nothing magical about this; it's just another way to write vectors that you might need to know. All the operations still work exactly the same. For example:

let $\mathbf{u} = 3\mathbf{i} + 2\mathbf{j} + 3\mathbf{k}$

let $\mathbf{v} = -3\mathbf{i} - 5\mathbf{j} + 12\mathbf{k}$

$$\mathbf{u} + \mathbf{v} = 3\mathbf{i} + 2\mathbf{j} + 3\mathbf{k} - 3\mathbf{i} - 5\mathbf{j} + 12\mathbf{k}$$
$$= 0\mathbf{i} - 3\mathbf{j} + 15\mathbf{k} = <0, -3, 15>$$

Matrices and Linear Algebra

3D graphics involve nothing more than performing mathematical operations over and over on lists of numbers; hence the use of matrices simplifies the process of transforming and representing this type of data.

A *matrix* is a rectangular array of numbers with a given number of rows and columns. We usually say that a matrix is *m*×*n,* meaning it has *m* rows and *n* columns. The size *m*×*n* is also referred to as the *dimension* of the matrix. For example, here's a matrix **A** that has dimension 2×2:

$$\mathbf{A} = \begin{vmatrix} 1 & 4 \\ 9 & -1 \end{vmatrix}$$

Notice that I use uppercase bold to denote the matrix. In general, most texts use uppercase bold to denote matrices, but I have seen uppercase italics, too. However, in this book we will use uppercase bold. In the example preceding, the first row is <1 4>, and the second row is <9 –1>. Here's a 3×2 matrix:

$$\mathbf{B} = \begin{vmatrix} 5 & 6 \\ 2 & 3 \\ 100 & -7 \end{vmatrix}$$

Playrix Ex. 1023, page 298

07 0672318350 CH04 05•02•2003 2:35 PM Page 274

And here's a 2×3 matrix:

```
C = ¦ 3 5 0  ¦
    ¦-8 12 4 ¦
```

To locate the <i,j>th element in the matrix, you simply look at the *i*th row and the *j*th column and look up the value. However, there is a gotcha—most math books start counting off matrix elements using 1 as the starting point, rather than 0 as we do in computer programs, so keep that in mind. We're going to start counting with 0 because this will make using C/C++ matrices work more naturally. For example, here's the labeling of a 3×3 matrix:

$$\mathbf{A} = \begin{¦}a_{00} & a_{01} & a_{02}\\ a_{10} & a_{11} & a_{12}\\ a_{20} & a_{21} & a_{22}\end{¦}$$

Easy enough. So that's all there is to the actual matrix itself and the labeling conventions, but you might ask, "Where do matrices come from?" Matrices are simply mathematical tools to represent systems of equations. Basically, if you have a system of equations like

```
3*x + 2*y = 1
4*x - 9*y = 9
```

that's a lot of work writing all the variables down. You know that the equation is in terms of (x,y), so why keep writing them? Why not just create a compact format that contains only the stuff you want to work with? This is how matrices came to be. In the preceding example, there are three different sets of values that we can dump into matrices, and we can work with them together or independently.

1. The coefficients of the variables:

   ```
   3*x + 2*y = 1
   4*x - 9*y = 9
   A = ¦3  2¦
       ¦4 -9¦
   ```

   Note: Dimension of **A** is 2×2.

2. The variables themselves:

   ```
   3*x + 2*y = 1
   4*x - 9*y = 9
   X = ¦x¦
       ¦y¦
   ```

   Note: Dimension of **X** is 2×1.

Playrix Ex. 1023, page 299

07 0672318350 CH04 05•02•2003 2:35 PM Page 275

3. And finally, the constants to the right:

```
3*x + 2*y = 1
4*x - 9*y = 9

B = ¦1¦
    ¦9¦
```

Note: Dimension of **B** is 2×1.

With all these nice matrices, we can focus on the coefficient matrix **A**, for example, without all the other stuff. Moreover, we can write matrix equations like

```
A*X = B
```


which, if you perform the math, (the multiplication), gives:

```
3*x + 2*y = 1
4*x - 9*y = 9
```

But how to perform the math? We learned that multiplication with vectors is non-intuitive, so what's the operation with matrices? Let's take a look by first understanding the concept of "1" in the context of matrices.

The Identity Matrix

The first thing we need to define in any mathematical system is 1 and 0. In matrix mathematics, there are analogs of both of these values. The analog of 1 is called the *identity matrix* **I**, and is created by placing all ones in the main diagonal of the matrix and zeros everywhere else. The identity matrix has the property:

```
A*I = I*A = A
```

Note: **I** and **A** must be the same size.

Furthermore, because matrices can be any size, there is obviously an infinite number of identity matrices, but there is one constraint—all identity matrices must be *square*, or in other words, *m*×*m*, where *m* >= 1. Here are a couple of examples:

```
I₂ =   ¦1 0¦
       ¦0 1¦
```

Dimension 2×2

```
       ¦1 0 0¦
I₃ =   ¦0 1 0¦
       ¦0 0 1¦
```

Dimension 3×3

Playrix Ex. 1023, page 300

07 0672318350 CH04 05•02•2003 2:35 PM Page 276

Technically, the identity matrix isn't exactly the analog of 1, but is under matrix multiplication (which we'll get to in a moment).

The second type of fundamental matrix is called the *zero matrix* **Z**, and is 0 under both addition and multiplication. It's nothing more than a matrix of dimension *m*×*n* with all entries 0. Other than that, there are no other special constraints.

$$\mathbf{Z}_{3x3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{Z}_{1x2} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

The only interesting thing about the zero matrix is that it has the standard properties of scalar 0 for both matrix addition and multiplication. Other than that, it's pretty useless.

Matrix Addition

Addition and subtraction of matrices is performed by adding or subtracting each element in two matrices to arrive at the result for each entry. The only rule to addition and subtraction is that the matrices that the operation is being performed on must be the same dimension. Here are two examples:

Let

$$\mathbf{A} = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix}$$

then

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} + \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix} = \begin{bmatrix} (1+13) & (5+7) \\ (-2+5) & (0-10) \end{bmatrix} = \begin{bmatrix} 14 & 12 \\ 3 & -10 \end{bmatrix}$$

and

$$\mathbf{A} - \mathbf{B} = \begin{bmatrix} 1 & 5 \\ -2 & 0 \end{bmatrix} + \begin{bmatrix} 13 & 7 \\ 5 & -10 \end{bmatrix} = \begin{bmatrix} (1-13) & (5-7) \\ (-2-5) & (0-(-10)) \end{bmatrix} = \begin{bmatrix} -12 & -2 \\ -7 & 10 \end{bmatrix}$$

Transpose of a Matrix

Sometimes it's handy to be able to represent a matrix with its rows swapped with its columns. For example, for the 1×3 row matrix $\mathbf{M}$ = [x y z], the transpose $\mathbf{M}^t$ (or [x y z]^t) really means a 3×1 column matrix that looks like this:

$$\mathbf{M}^t = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Playrix Ex. 1023, page 301

07 0672318350 CH04 05•02•2003 2:35 PM Page 277

For the general case of an *m*×*n* matrix, the transpose is computed simply by exchanging the rows with the columns. For example, for the 3×2 matrix **A** shown here:

```
      ¦5  6  ¦
A =   ¦2  3  ¦
      ¦100 -7¦
```

The transpose is 2×3:

```
       ¦5  2    100¦
A^t =  ¦6   3  -7  ¦
```

That's all there is to it. When computing inverses and performing general matrix multiplications, the transpose of a matrix is sometimes employed.

Matrix Multiplication

There are two forms of matrix multiplication: *scalar-matrix* and *matrix-matrix*. Scalar matrix multiplication is simply the multiplication of a matrix by a scalar number. You simply multiply each element of the matrix by the number. The matrix can be any size *m*×*n*. Here's a general description for a 3×3 matrix:

Let k be any real constant.

$$\text{Let } \mathbf{A} = \begin{¦a_{00} & a_{01} & a_{02}¦ \\ ¦a_{10} & a_{11} & a_{12}¦ \\ ¦a_{20} & a_{21} & a_{22}¦\end{}$$

$$\text{Then } k*\mathbf{A} = k*\begin{¦a_{00} & a_{01} & a_{02}¦ \\ ¦a_{10} & a_{11} & a_{12}¦ \\ ¦a_{20} & a_{21} & a_{22}¦\end{} = \begin{¦k*a_{00} & k*a_{01} & k*a_{02}¦ \\ ¦k*a_{10} & k*a_{11} & k*a_{12}¦ \\ ¦k*a_{20} & k*a_{21} & k*a_{22}¦\end{}$$

Here's an example:

```
3*¦ 1 4¦ = ¦(3*1)  (3*4)¦ = ¦3 12¦
 ¦-2 6¦  ¦(3*(-2)) (3*6)¦  ¦-6 18¦
```

MATH

Scalar multiplication is also valid for matrix equations, as long as you perform the multiplication to both sides. This is true because you can always multiply the coefficients of any system of equations by a constant as long as you do so to both the right-hand side and the left-hand side of the system.

The second form is true matrix multiplication. Its mathematical basis is a bit complex, but more or less you can think of a matrix as an "operator" that operates on another matrix.

Playrix Ex. 1023, page 302

07 0672318350 CH04 05•02•2003 2:36 PM Page 278

Anyway, any two matrices **A** and **B** that you want to multiply must have the same inner dimension, or in other words:

*If **A** is m×n, then **B** must be n×r.*

m and r may or may not be equal, but the inner dimension must. For example, you can multiply a 2×**2** by a **2**×2, a 3×**2** by a **2**×3, and a 4×**4** by a **4**×5, but you can't multiply a 3×**3** by a **2**×4 because the inner dimensions 3 and 2 are not equal. The resulting matrix of the multiplication will have a size that is equal to the outer dimension of the multiplier and multiplicand matrix. For example, a 2×3 multiplying a 3×4 would have dimension 2×4.

Matrix multiplication is one of those things that's very hard to describe with words, so take a look at Figure 4.25 while I give you the technical description of the multiplication algorithm.

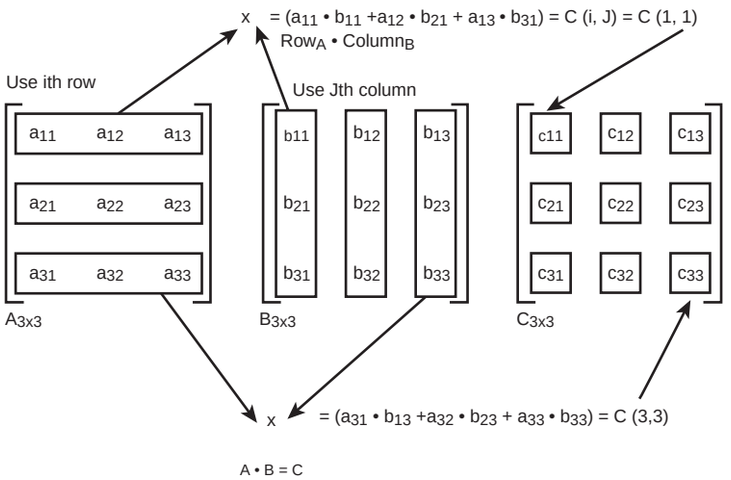


FIGURE 4.25 The mechanics of matrix multiplication.

Given a matrix **A** and **B**, to multiply them together (**A** × **B** = **C**) to compute each element of the resulting matrix **C**, you must take a row of **A** and multiply it by a column in **B**. To perform the multiplication, you will sum the products of each element or take the dot product. Here's an example for a 2×2 multiplying a 2×3—order counts!

```
Let A = ¦1 2¦   B = ¦1 3 5¦
        ¦3 4¦       ¦6 0 4¦

C = A x B = ¦(1*1 + 2*6) (1*3 + 2*0) (1*5 +2*4)¦
            ¦(3*1 + 4*6) (3*3 + 4*0) (3*5 +4*4)¦

= ¦13 3 13¦
  ¦27 9 31¦
```

Playrix Ex. 1023, page 303

07 0672318350 CH04 05•02•2003 2:36 PM Page 279

TIP

As an aside, I want to bring to your attention the bolded sum of products (1*1 + 2*6). This, and all the other elements of **C**, are really vector dot products of each row of **A** and each column of **B**. Hence, the $\mathbf{C}_{ij}$th element can also be computed as the ith row of **A** dotted with the jth column of **B**.

So that's how you multiply matrices. Another way to think of it is, if you want to compute the product of $\mathbf{A} \times \mathbf{B}$, call it **C**. You can do this element by element. Hence, if you want the c_{ijth} element (where both i and j are zero-based), you can find it by taking the ith row of **A** and dotting (summing the products) with the jth column of **B**.

Laws for Matrix Operations

Just as the dot product and cross product have specific laws relating to addition and multiplication in the context of the associative, commutative, and distributive laws, so do matrices. Here they are:

(a) **A** + **B** = **B** + **A** (Commutative law for addition)

(b) **A** + (**B** + **C**) = (**A** + **B**) + **C** (Associative law for addition)

(c) **A***(**B***C**) = (**A***B**)*C** (Associative law for multiplication)

(d) **A***(**B**+**C**) = **A***B** + **A***C** (Distributive law)

(e) k*(**A** + **B**) = k***A** + k***B** (Distributive law)

(f) (**A**+**B**)***C** = **A***C** + **B***C** (Distributive law)

(g) **A***I** = **I***A** = **A** (Multiplicative property of identity)

Note: All the "+"s can be replaced with "-"s.

However, the following is not true in general:

$$(\mathbf{A}*\mathbf{B}) \neq (\mathbf{B}*\mathbf{A})$$

That is, the commutative law of multiplication does not hold for matrices. Thus, the order in which matrices are multiplied is very important.

NOTE

One case in which (**A***B**) does equal (**B***A**) is if **A** or **B** is equal to the identity matrix.

Computing the Inverse and Solving Systems

At this point, you know how to create and manipulate matrices, but there is one more mathematical object we need to learn to compute—the *multiplicative inverse*. In other

Playrix Ex. 1023, page 304

07 0672318350 CH04 05•02•2003 2:36 PM Page 280

words, if you have a scalar *x* and you want to find another number x^{-1} such that $x*x^{-1} = 1$, then x^{-1} is the multiplicative inverse, or simply the *inverse*. In the case of scalars, it's easy. Given x is a real number, x^{-1} can be computed as follows:

$x^{-1} = 1/x$, for all $x \neq 0$.

That's pretty basic math. However, finding the multiplicative inverse for a matrix is rather complex. For example, in the matrix equation, we want to solve for **X**:

$$\mathbf{A}*\mathbf{X} = \mathbf{B}$$

If we had the inverse of **A** ($\mathbf{A}^{-1}$), we could multiply it by both sides:

$$(\mathbf{A}^{-1}*\mathbf{A})*\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

And because we know that anything multiplied by its inverse is the identity matrix, we have

$$(\mathbf{I})*\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

And any matrix multiplied by the identity matrix is that matrix, thus we have the solution for **X**:

$$\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

Being able to solve systems of equations this way is very important in many areas of 3D game programming, such as graphics, physics, and animation. We don't really have time to cover how to find the inverse of a matrix **A** in general, but I will give you some starting places and show you how for the 2×2 case.

Given the matrix:

$$\mathbf{A} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

The inverse exists, and there is a unique solution to the system if the *determinate* of the matrix, that is, Det(**A**), is not zero. The determinate of an **A** (or any 2×2 matrix) can be computed as the following:

EQUATION 4.16 Determinate of a 2×2 Matrix

$$\text{Det}(\mathbf{A}) = (a*d - b*c)$$

If this result is not zero, the inverse of **A**, $\mathbf{A}^{-1}$ exists and can be computed as follows:

Playrix Ex. 1023, page 305

07 0672318350 CH04 05•02•2003 2:36 PM Page 281

EQUATION 4.17 Inverse of a 2×2 Matrix

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} * \begin{vmatrix} d & -b \\ -c & a \end{vmatrix}$$

Multiplying this all out, we get:

$$\mathbf{A}^{-1} = \begin{vmatrix} d/(a*d - b*c) & -b/(a*d - b*c) \\ -c/(a*d - b*c) & a/(a*d - b*c) \end{vmatrix}$$

Let's put this to the test in a real system of equations. Assume you have the following two linear equations:

```
 3*x+5*y = 6
-2*x+2*y = 4
```


We want to solve for x,y using matrix techniques, so we first extract out the coefficient matrix **A**, the variable matrix X, and the constant matrix **B**:

$$\mathbf{A} = \begin{vmatrix} 3 & 5 \\ -2 & 2 \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

$$\mathbf{X} = \begin{vmatrix} x \\ y \end{vmatrix}$$

$$\mathbf{B} = \begin{vmatrix} 6 \\ 4 \end{vmatrix}$$

Therefore, we want to solve for X in

$$\mathbf{A}*\mathbf{X} = \mathbf{B},\ \mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

Computing the inverse of **A**, we get:

$$\mathbf{A} = \begin{vmatrix} 3 & 5 \\ -2 & 2 \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

$$(a*d - b*c) = (3*2 - (-2)*5) = (6+10) = 16$$

$$\mathbf{A}^{-1} = \begin{vmatrix} d/(a*d - b*c) & -b/(a*d - b*c) \\ -c/(a*d - b*c) & a/(a*d - b*c) \end{vmatrix}$$

$$\mathbf{A}^{-1} = \begin{vmatrix} 2/16 & -5/16 \\ 2/16 & 3/16 \end{vmatrix}$$

Playrix Ex. 1023, page 306

07 0672318350 CH04 05•02•2003 2:36 PM Page 282

Plugging into the matrix equation:

```
X = A-1*B

X =   ¦2/16 -5/16¦ * ¦6¦ = ¦(2/16)*6 +(4)*(-5)/16 ¦
      ¦2/16  3/16¦   ¦4¦   ¦(2/16)*6 +  (4)*3/16  ¦

      = ¦12/16 - 20/16 ¦
        ¦12/16 + 12/16 ¦

      = ¦-8/16¦ = ¦-0.5¦
        ¦24/16¦   ¦ 1.5¦
```

Therefore, x = -0.5 and y = 1.5. Let's plug these values into the original system and see what's up:

```
   3*x  +   5*y     = 6
3*(-0.5)+   5*(1.5) = 6
 -1.5   +   7.5     = 6
```

That's correct. And just to make sure, let's check the second equation:

```
-2*x      +   2*y     = 4
-2*(-0.5) +   2*(1.5) = 4
   1.0    +         3  = 4
```

Check-check! Cool, huh?

Cramer's Rule

The problem with solving a large system of equations that has three or more variables using Cramer's Rule is that the technique is bottlenecked by the computation needed to find the inverse of the coefficient matrix. This can be rather complex, and usually must be done iteratively for systems that have more than three variables by using techniques such as *Gaussian elimination*.

NOTE

Gaussian elimination is a process whereby a coefficient matrix is transformed into a form such that the solution to the system can literally be read off. The process is based on the fact that any row of a matrix can be multiplied by a number, or a multiple of another row can be added to any row as long as the same is done to the constant matrix. Using these basic rules, one can slowly convert a matrix into a form where only one non-zero coefficient in a row exists, which translates to a solved variable. Then the process can be continued, or you can employ back substitution to solve the rest of the system.

Playrix Ex. 1023, page 307

07 0672318350 CH04 05•02•2003 2:36 PM Page 283

However, there is a closed form solution for a three-variable system based on *Cramer's Rule*, which you have probably heard of before. Cramer's Rule for a three-variable system is as follows:

Given a system:

$$a_{00}*x + a_{01}*y + a_{02}*z = b_0$$
$$a_{10}*x + a_{11}*y + a_{12}*z = b_1$$
$$a_{20}*x + a_{21}*y + a_{22}*z = b_2$$

We can convert it to matrix form as follows:

$$\mathbf{A} = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

$$\mathbf{X} = \begin{vmatrix} x \\ y \\ z \end{vmatrix}$$

$$\mathbf{B} = \begin{vmatrix} b_0 \\ b_1 \\ b_2 \end{vmatrix}$$

At this point, we must compute the determinate of **A**, or Det(**A**), which for a 3×3 matrix is generally

$$\mathbf{A} = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

EQUATION 4.18a Determinate of a 3×3 Matrix

$$Det(\mathbf{A}) = a_{00}*a_{11}*a_{22} + a_{01}*a_{12}*a_{20} + a_{02}*a_{10}*a_{21} - a_{02}*a_{11}*a_{20} - a_{01}*a_{10}*a_{22} - a_{00}*a_{12}*a_{21}$$

Note: 12 multiplications and 5 additions.

There is also another method of computing the determinate of a matrix based on *cofactor expansion*. This is rather advanced, so I will just give you the result:

EQUATION 4.18b Determinate of a 3×3 Matrix Using Cofactors

$$Det(\mathbf{A}) = a_{00}*(a_{11}*a_{22} - a_{21}*a_{12}) - a_{01}*(a_{10}*a_{22} - a_{20}*a_{12}) + a_{02}*(a_{10}*a_{21} - a_{20}*a_{11})$$

Note: 9 multiplications and 5 additions.

Playrix Ex. 1023, page 308

07 0672318350 CH04 05•02•2003 2:36 PM Page 284

And just to be complete with the general notation, the determinate of a 2×2 matrix in general form is

$$\mathbf{A} = \begin{vmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{vmatrix}$$

$$\text{Det}(\mathbf{A}) = a_{00} * a_{11} - a_{01} * a_{10}$$

Now that we have the Det(**A**) for the 3×3 system, Cramer's Rule can be written as a set of quotients that solve for x,y, and z. There's one quotient for each solution variable; x,y,z, where the numerator is the determinate of the coefficient matrix **A**, with the first, second, and third columns replaced with the column vector **B** to compute x,y, and z, respectively. Then the denominator of each quotient is simply Det(**A**). Or more specifically, we have

$$\mathbf{x} = \frac{\text{Det}\begin{vmatrix} b_0 & a_{01} & a_{02} \\ b_1 & a_{11} & a_{12} \\ b_2 & a_{21} & a_{22} \end{vmatrix}}{\text{Det}(\mathbf{A})}$$

Note: The first column of **A** was replaced with **B** to solve for x.

$$\mathbf{y} = \frac{\text{Det}\begin{vmatrix} a_{00} & b_0 & a_{02} \\ a_{10} & b_1 & a_{12} \\ a_{20} & b_2 & a_{22} \end{vmatrix}}{\text{Det}(\mathbf{A})}$$

Note: The second column of **A** was replaced with **B** to solve for y.

$$\mathbf{z} = \frac{\text{Det}\begin{vmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \\ a_{20} & a_{21} & b_2 \end{vmatrix}}{\text{Det}(\mathbf{A})}$$

Note: The third column of **A** was replaced with **B** to solve for z.

And of course, you must use one of the forms of Equation 4.18 to compute the determinate of each numerator, but you get the idea. Later we will write software to compute determinates and solve for systems of two and three variables.

Playrix Ex. 1023, page 309

07 0672318350 CH04 05•02•2003 2:36 PM Page 285

Transformations Using Matrices

The whole point of learning all this stuff about matrices is to use them to perform 2D/3D transformations. Basically, what we're going to do is multiply the 2D/3D point **p** that we want transformed against the desired transformation matrix. Or mathematically:

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}$$

where **p**' is the transformed point, **p** is the original point, and **M** is the transformation matrix. Additionally, we can perform as many transformations as we want by multiplying transformation matrices together. That is, we can *concatenate* any number of matrices together by pre-multiplying them in a series, like this:

$$\mathbf{M} = \mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*\ldots\mathbf{M}_n$$

The following transforms are equivalent:

$$\mathbf{p}' = \mathbf{p}*\mathbf{M} = \mathbf{p}*\mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*\ldots\mathbf{M}_n$$

Whether you multiply **p** by **M** or the product $\mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*\ldots\mathbf{M}_n$, the results are the same. For example, say that we have three transformations $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$ that we want to apply to the point **p** in order to arrive at **p**'. These could represent a translation, rotation, and projection, for example. We could perform this series of transforms in the normal way, one at a time:

$$\mathbf{p1} = \mathbf{p}*M_1$$

$$\mathbf{p2} = \mathbf{p1}*M_2$$

$$\mathbf{p}' = \mathbf{p2}*M_3$$

Note: **p1** and **p2** are just temporary variables.

The preceding series of transforms is completely acceptable. But, what if we have thousands of **p**'s to transform, or even millions? It seems like those intermediate steps are a waste—and they are! We can concatenate the transforms together like this, via substitution:

$$p1 = p*M_1$$

$$p2 = p*M_1*M_2$$

$$p' = p*(M_1*M_2*M_3)$$

Playrix Ex. 1023, page 310

07 0672318350 CH04 05•02•2003 2:36 PM Page 286

Hence, the transform to **p** can be represented by the transform **p** *(**M**), where $\mathbf{M}_{123} = (\mathbf{M}_1 * \mathbf{M}_2 * \mathbf{M}_3)$. Therefore, we can compute $\mathbf{M}_{123} = (\mathbf{M}_1 * \mathbf{M}_2 * \mathbf{M}_3)$ once and then use it instead of saving two matrix multiplies per point! Thus, the transform becomes

$M_{123} = (M_1 * M_2 * M_3)$

$p' = p * M_{123}$

Of course, we have to compute $\mathbf{M}_{123}$, but this has to be done only once, and then we can run as many points through the transform as we wish. The process of concatenating matrices together is very useful in the 3D pipeline, especially when transforming 3D vertices in object space into final points in screen space (but we'll get to that).

NOTE

Because of the noncommutative nature of matrix multiplication in the general case, order is very important, and $\mathbf{M}_1 * \mathbf{M}_2 * \mathbf{M}_3 \neq \mathbf{M}_3 * \mathbf{M}_2 * \mathbf{M}_1$.

Homogeneous Coordinates

The next topic I want to cover briefly is *homogeneous coordinates*. In most cases, we're going represent our 3D objects as lists of vertices, each having three components: **p**(x,y,z), or as a vector <x, y, z>, or as a row matrix [x y z]—note that all are equivalent and the difference is really notational. I will use the latter when I want to multiply with a matrix and/or I don't want to use commas. However, to be able to use the full power of matrix transformations, we are going to represent points in 2D and 3D using homogeneous coordinates, so we can perform all possible transformations using matrices. To convert a 2D or 3D point to homogeneous coordinates, we add a "w" term to each, as follows:

The homogeneous coordinate for a 3D point [x y z]:

```
[x y z w]
```

and for a 2D point [x y]:

```
[x y w]
```

w homogenizes the coordinate, but it's not artificial. To convert from homogenous coordinates to normal coordinates, we must divide by w. For example, in the 3D case:

Given

```
[x y z w]
```

the normal coordinate can be computed as

```
[x/w y/w z/w]
```

Playrix Ex. 1023, page 311

07 0672318350 CH04 05•02•2003 2:36 PM Page 287

Therefore, if w = 1, the transform doesn't change x, y, or z. So what's the point of adding w? Well, w allows us to represent the concept of infinity. For example, as w approaches 0, then x/w approaches ∞. Or in other words, as the denominator of any fraction goes to 0, the quotient goes to infinity. Additionally, homogenous coordinates come from *homogenous equations*. For example, take a look at the following linear equation:

```
a*x + b*y + c = 0
```

This is a linear equation with degree 1, but two of the terms have degree 1 while the constant term has degree 0. What if we want all the degrees to be the same (homogenous)? We can convert the polynomial to a homogenous one by replacing x with x/w and y with y/w:

```
a*x/w + b*y/w + c = 0
```


And then we want to clear the w's in the denominator, so we multiply through by w, and the result is

```
a*x + b*y + c*w = 0
```

Now all the terms have degree 1, and thus the polynomial is homogenous. The point of all this is that if you represent geometric objects using homogenous coordinates, you can guarantee that all transformations are possible, including perspective transforms and translations. I don't want to belabor this anymore, but know that we are going to use a dummy variable w = 1 in many cases just so we can represent all transforms via matrices. In reality, we might not use any storage for w in our data structures because we are going to let w = 1 for most cases, and all homogenous coordinates [x y z w] with w = 1 are simply [x y z]. I just want you to know where this mathematical trick comes from.

Although we are going to cover the full gamut of 3D transformations in the next chapter in great detail, I just want to briefly show some of the most basic 3D transformations (*translation*, *scaling*, and *rotation*) and why we need homogeneous coordinates to support all three.

Applied Matrix Transformations

The following section illustrates some of the basic matrix transformations that we will perform to both 2D and 3D data. The examples are for 3D, but the 2D versions can easily be derived by dropping the z component in most cases. Note that w = 1 for all the examples.

Playrix Ex. 1023, page 312

07 0672318350 CH04 05•02•2003 2:36 PM Page 288

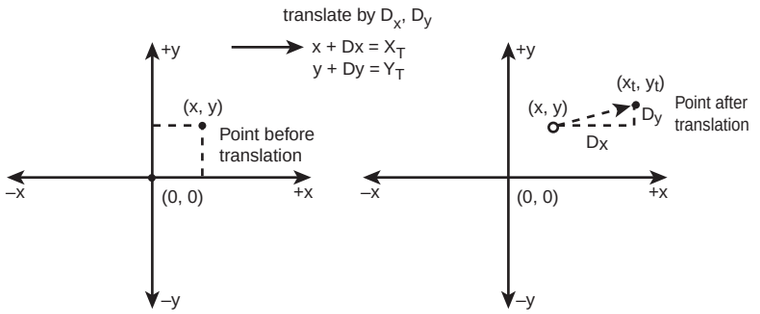


FIGURE 4.26 Translation depicted geometrically for the 2D case.

Translation

To perform translation in 3D, we want to translate the point **p**(x,y,z) to the new point **p**'(x+dx, y+dy, z+dz). This is shown in Figure 4.26 for the simplified case where the point is in the x-y plane. The following 4×4 matrix will do the job:

$$\mathbf{M}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

Given **p** = [x y z 1], we want to transform:

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}_t = [x\ y\ z\ 1]*\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

$$= [(x+1*dx)\ (y+1*dy)\ (z+1*dz)\ (1*1)]$$

$$= [(x+dx)\ (y+dy)\ (z+dz)\ 1]$$

And dividing the first three components (which represent x, y, and z) by the w factor (1.0), we get:

$$= [(x+dx)/1\ (y+dy)/1\ (z+dz)/1]$$

$$= \mathbf{p}'((x+dx),\ (y+dy),\ (z+dz))$$

NOTE

In reality, when we write the math library, we obviously aren't going to waste space and divide by 1.0 all the time. We are going to use a 4×3 matrix for most transforms and assume w = 1, but for now, let's just pretend.

Playrix Ex. 1023, page 313

07 0672318350 CH04 05•02•2003 2:36 PM Page 289

And if we pull out the first three elements, we get

```
x' = x+dx
y' = y+dy
z' = z+dz
```

which is exactly what we wanted.

Translation Inverse

Also, it's interesting to note that the inverse of $\mathbf{M}_t$ is computed simply by changing the sign of dx, dy, and dz in the matrix $\mathbf{M}_t$:

$$\mathbf{M}_t^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{bmatrix}$$

Let's check that by computing the product $\mathbf{M}_t * \mathbf{M}_t^{-1}$, which should be $\mathbf{I}_{4x4}$:

$$\mathbf{M}_t * \mathbf{M}_t^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_{4x4}.$$

NOTE

Computing the inverse efficiently for common transformation matrices is very important in 3D graphics, especially in the camera transform.

Scaling

To scale a point relative to the origin, we want to multiply the point **p**(x,y,z) components by scaling factors sx, sy, and sz for the x- ,y-, and z-axes, respectively. This is shown in Figure 4.27 for the simplified case where all the points are in the x-y plane. In addition, we want no translation during the scaling operation. Here's the matrix we want:

$$\mathbf{M}_s = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Playrix Ex. 1023, page 314

07 0672318350 CH04 05•02•2003 2:36 PM Page 290

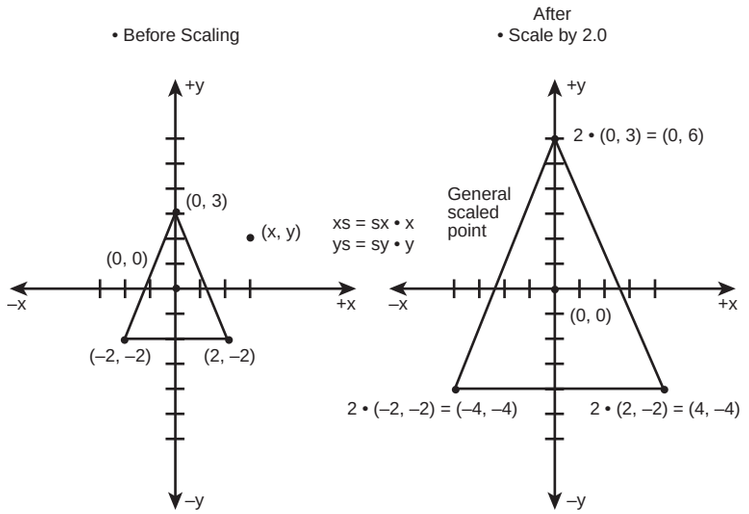


FIGURE 4.27 The scaling operation.

Given **p** = [x y z 1], we want to transform

$$\mathbf{p}' = \mathbf{p}^* \mathbf{M}_s = [x\ y\ z\ 1] * \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
= [(x*sx) (y*sy) (z*sz) (1*1)]

  = [(x*sx) (y*sy) (z*sz) 1]
```

And dividing the first three components (which represent x, y, z) by the w factor, we get

```
= [(x*sx)/1 (y*sy)/1 (z*sz)/1]

= p'((x*sx), (y*sy), (z*sz))
```

which again is the desired result for scaling; that is

```
x' = sx*x
y' = sy*y
z' = sz*z
```

Playrix Ex. 1023, page 315

07 0672318350 CH04 05•02•2003 2:36 PM Page 291

MATH

Note the 1 in the lower-right bottom of the transformation matrix. Technically, it's not necessary because we aren't ever going to use the result from the fourth column for scaling. Hence, we are wasting math cycles. Again, as noted in the translation example, using the four-component representation to implement homogenous coordinates in our 3D points (or three-component for 2D points) forces us to use a 4×4 matrix to be mathematically correct. Later we will throw away all that we don't need.

Scaling Inverse

Although not as commonly used as the inverse of translation, computing the inverse of the scaling matrix can come in handy. Can you think of what the inverse is? Give up? The inverse of the scaling matrix $\mathbf{M}_s$ is computed simply by inverting each of the scaling factors sx, sy, and sz in the matrix $\mathbf{M}_s$:

$$\mathbf{M}_s^{-1} = \begin{bmatrix} 1/sx & 0 & 0 & 0 \\ 0 & 1/sy & 0 & 0 \\ 0 & 0 & 1/sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let's check that by computing the product $\mathbf{M}_s * \mathbf{M}_s^{-1}$, which should be $\mathbf{I}_{4x4}$:

$$\mathbf{M}_s * \mathbf{M}_s^{-1} =$$

$$= \underbrace{\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{M}_s} * \underbrace{\begin{bmatrix} 1/sx & 0 & 0 & 0 \\ 0 & 1/sy & 0 & 0 \\ 0 & 0 & 1/sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{M}_s^{-1}}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_{4x4}.$$

NOTE

Note that computing the inverse mathematically using Gaussian elimination or another method is very complex and tedious, whereas here we simply thought about the problem in terms of its geometrical properties to find the inverse. In general, when a transform represents a geometrical transformation in 2D or 3D, computing its inverse is trivial, and can usually be done with a little thought about simply inverting the geometrical transformation. Keep this in mind when we compute the inverses for the rotation matrices and see whether you can figure them out before getting to that section.

Playrix Ex. 1023, page 316

07 0672318350 CH04 05•02•2003 2:36 PM Page 292

Rotation

The rotation matrix is the most complex of all the transformations because it's full of trig functions. Basically, we want to rotate the input point by using the rotation equations. To achieve this, we must look at the rotation equations and pick off the operators and then push them into a matrix. In addition, we don't want any translation, so the bottom row in positions 0, 1, and 2 will always be 0.

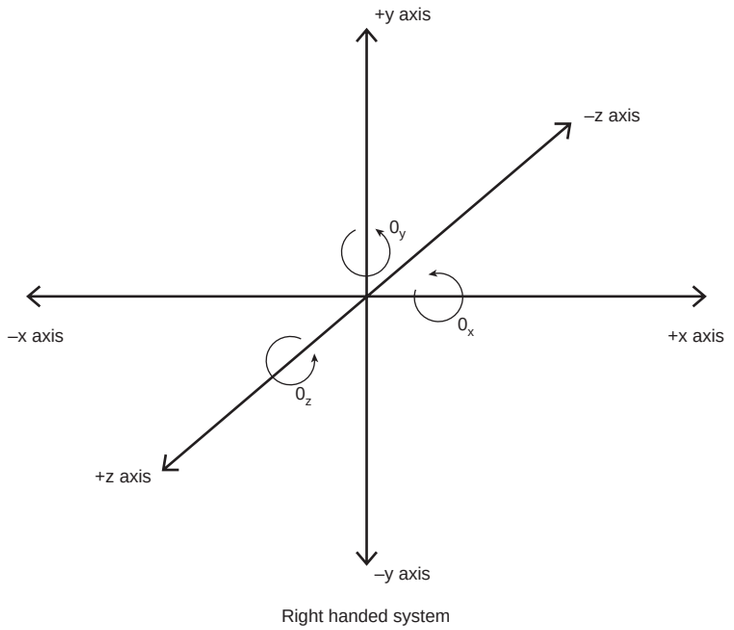


FIGURE 4.28 The axes of rotation.

There is also one more technical detail—in a 3D system, there are three axes that we can rotate in parallel to: the x, y, or z, as shown in Figure 4.28. With that in mind, we can start with the z-axis rotation and then use it as a model to create the x- and y-axis rotation matrices. The key is that during rotation around any axis, the axis component itself remains unchanged.

CAUTION

The direction of rotation for positive values of θ depends on the "handedness" of the system. If you are using a right-handed system, all rotations with positive θ will be counterclockwise. If you are using a left-handed system, positive θ will result in clockwise rotations.

Playrix Ex. 1023, page 317

07 0672318350 CH04 05•02•2003 2:36 PM Page 293

Keeping all that in mind, here are the rotation equations for rotation parallel to the x-, y-, and z-axes:

Rotation parallel to the z-axis:

$$\mathbf{M}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

NOTE

If you delete the last column and row, you'll have the matrix for 2D rotation in a plane.

Rotation parallel to the x-axis:

$$\mathbf{M}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation parallel to the y-axis:

$$\mathbf{M}_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, let's look at an example for rotation around the z-axis and see whether the transform works.

Given the point $\mathbf{p}$ = [x y z 1], we want to rotate an angle θ about the z-axis:

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}_z = [x\ y\ z\ 1] * \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
p' = [(x*cos θ – y*sin θ) (x*sin θ + y*cos θ) z (1*1)]
```

And dividing the first three components (which represent x, y, and z) by the w factor, we get:

```
p' = ((x*cos θ–y*sin θ), (x*sin θ + y*cos θ), z)
```

Note: z remained unchanged, as it should.

Playrix Ex. 1023, page 318

07 0672318350 CH04 05•02•2003 2:36 PM Page 294

Rotation Inverse

The rotation inverses for $\mathbf{M}_x$, $\mathbf{M}_y$, and $\mathbf{M}_z$ are all computed in the same manner, so I'm only going to show you the derivation of $\mathbf{M}_z$—the others will follow from there. To compute the inverse of $\mathbf{M}_z$ for a given θ, you can take two approaches: one based on geometry, and the other based on linear algebra. Let's begin with the geometric computation of the inverse. It makes sense that if we rotate an object parallel to the z-axis by θ, then to rotate it back, we can simply rotate it by -θ. That's it! So all you need to do is take the rotation matrix and replace θ with -θ, and that's the inverse. Therefore, the inverse $\mathbf{M}_z^{-1}$ can be computed as follows:

$$\mathbf{M}_z^{-1} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) & 0 & 0 \\ -\sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, there's another simplification we can make here to get rid of the -θ terms. Using the *reflection identities* from Table 4.5, we have

$$\sin(-\theta) = -\sin(\theta)$$
$$\cos(-\theta) = \cos(\theta)$$

Using these formulas and rewriting $\mathbf{M}_z^{-1}$ in terms of positive θ, we have

$$\mathbf{M}_z^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let's try the product $\mathbf{M}_z * \mathbf{M}_z^{-1}$ and see whether we get the identity matrix $\mathbf{I}_{4x4}$:

$$\mathbf{M}_z * \mathbf{M}_z^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} (\cos^2\theta+\sin^2\theta) & (-\sin\theta*\cos\theta+\cos\theta*\sin\theta) & (0) & (0) \\ (-\sin\theta*\cos\theta+\cos\theta*\sin\theta) & (\sin^2\theta+\cos^2\theta) & (0) & (0) \\ (0) & (0) & (1) & (0) \\ (0) & (0) & (0) & (1) \end{bmatrix}$$

Playrix Ex. 1023, page 319

07 0672318350 CH04 05•02•2003 2:36 PM Page 295

Which, using the identity $\cos^2\theta+\sin^2\theta = 1$, reduces to

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_{4x4}.$$

See how powerful geometric interpretation is to compute inverses of geometry transforming matrices? Very cool! Anyway, now let's get a little mathematical. The rotation matrix itself is a special matrix in the wide world of matrices. In fact, it's part of a very unique class of matrices that are called orthonormal.

Orthonormal Matrices

Orthonormal matrices are matrices where each row or column is orthogonal to the previous for a given basis. A *basis* is a set of vectors that can represent all vectors in a space. In our case, the basis for standard 2D space is <i+j>, and the basis for 3D is <i+j+k>—remember this from a few sections ago? Remember, all vectors in a 3D space can be written as a linear combination:

$$\mathbf{u} = u_x*\mathbf{i} + u_y*\mathbf{j} + u_z*\mathbf{k}$$

Where **i** = <1,0,0>, **j** = <0,1,0>, and **k** = <0,0,1>. Therefore, i is orthogonal to j is orthogonal to k because for the set s1 = (**i**,**j**,**k**) with k = 0,1,2, and s2 = (**i**,**j**,**k**) with t = 0,1,2 and $k \neq t$, the dot product is 0; therefore, the vectors are perpendicular. For example, (**i**.**j**) equals:

```
<1,0,0> . <0,1,0> = 1*0 + 0*1 + 0*0 = 0.
```

This will hold true for all pairs of (**i**,**j**,**k**). Now, the point is that if a matrix contains a set of orthogonal vectors as its rows or columns, the matrix is said to be an orthogonal or orthonormal matrix. Let's take a look at $\mathbf{M}_z$ again:

$$\mathbf{M}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we assume that the rows represent vectors in 3D, we have:

```
U1 = <cos θ, sin θ, 0, 0>
U2 = <-sin θ, cos θ, 0, 0>
U3 = <0, 0, 1, 0>
U4 = <0, 0, 0, 1>
```

Playrix Ex. 1023, page 320

07 0672318350 CH04 05•02•2003 2:36 PM Page 296

Now, technically we are in 4D space because these transforms are based on homogeneous coordinates, but let's not worry about that for a moment and see whether all these vectors are orthogonal to one another.

- **u1** is orthogonal to **u4** because (**u1** . **u4**) = 0.
- It's easy to see that both **u1** and **u4** are orthogonal to both **u1** and **u2** because both **u1** and **u2** have zeros in the last two components, thus the products with either **u3** or **u4** will always be zero, and so will the dot product.

The only big question left is whether **u1** is orthogonal to **u2**. Let's see:

```
u1 = <cos θ, sin θ, 0, 0>
u2 = <-sin θ, cos θ, 0, 0>

u1 . u2 = (cos θ)*(-sin θ) + (sin θ)*(cos θ) + 0 + 0
    = 0.
```

Killer! The point is, we have shown what an orthogonal matrix is and how to check for it. Wonderful, but how does this help us find the inverse? Well, if a matrix is orthogonal, its transpose is its inverse. Therefore, if we compute $\mathbf{M}_z^t$, it should be equivalent to the inverse of $\mathbf{M}_z$ (that is, $\mathbf{M}_z^{-1}$).

Given

$$\mathbf{M}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

its transpose is computed by exchanging each column with each row:

$$\mathbf{M}_z^t = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is indeed $\mathbf{M}_z^{-1}$! Amazing, huh? I scare myself sometimes <BG>.

Basic Geometric Entities

The next topic I want to discuss is basic geometric entities such as points, lines, planes, and surfaces, but with a focus on various mathematical representations and operations that are of interest in the world of 3D graphics. For example, all polygons are really planes with boundaries; therefore, being proficient at manipulating planes and plane equations is very useful.

Playrix Ex. 1023, page 321

07 0672318350 CH04 05•02•2003 2:36 PM Page 297

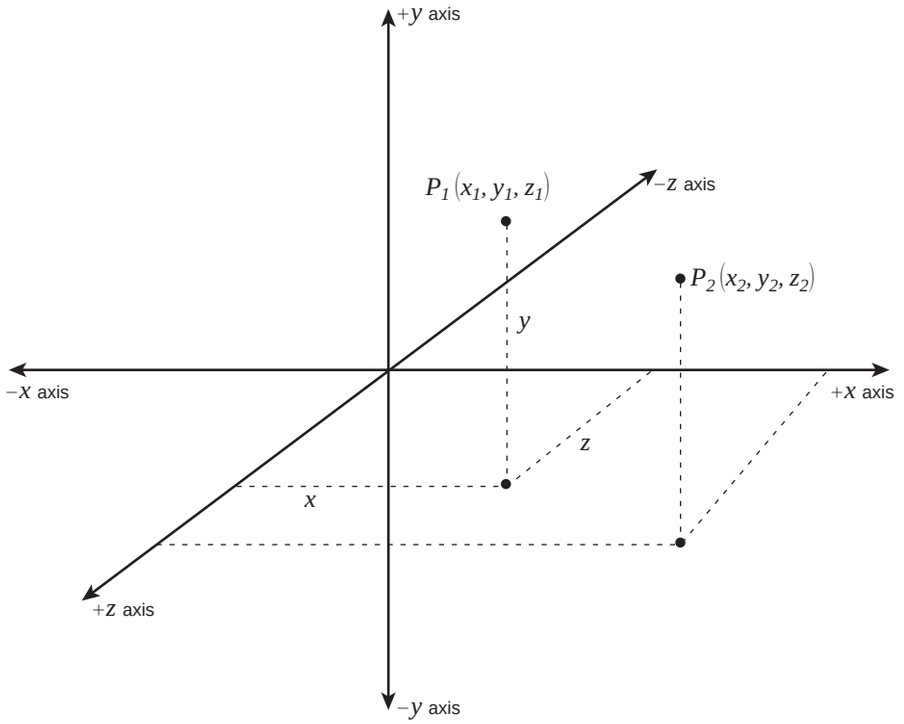


FIGURE 4.29 A plain old point in 3D.

Points

There's not much to say about a point. Figure 4.29 represents a point **p1**(x,y) in 2D and a point **p2**(x,y,z) in 3D. Of course, the homogeneous coordinates for **p1** and **p2** are [x y w] and [x y z w], respectively.

Lines

Lines are a bit more interesting, and a lot more complex to represent in 3D. Let's begin by looking at a line from **p1**(x1, y1) to **p2**(x2,y2) in 2D, as shown in Figure 4.30. Nothing special about this line. We can represent it in a number of ways, such as the *point-slope form*, the *slope-intercept form*, or the *general form*.

EQUATION 4.19 Slope-Intercept Form

```
y=m*x + b
```

where m is the slope of the line (dy/dx), and b is the y-axis intercept.

Playrix Ex. 1023, page 322

07 0672318350 CH04 05•02•2003 2:36 PM Page 298

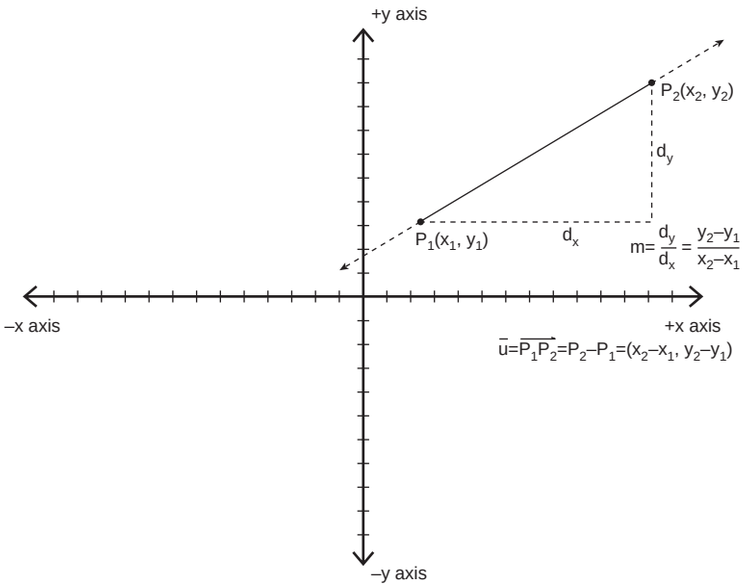


FIGURE 4.30 A line in 2D.

EQUATION 4.20 Point-Slope Form

```
(x-x0) = m*(y-y0)
```

where m is the slope and (x0,y0) is a point on the line.

EQUATION 4.21 General Form

```
a*x + b*y + c = 0
```

The general form, of course, is the most basic. Let's see whether we can convert it into the other forms. Converting to the slope-intercept form is trivial:

```
y = m*x + b
```

Rearranging, we get

```
m*x – 1*y + b = 0
```

We can now pick off a, b, and c in the general form:

```
a*x + b*y + c = 0
```

Playrix Ex. 1023, page 323

07 0672318350 CH04 05•02•2003 2:36 PM Page 299

Therefore, a = m, b = -1, and c = b (the intercept). But what if you don't have the intercept? How can we find the values of (a,b,c) if we only have the coordinates (x0,y0) and (x1,y1)? Well, let's see whether the point-slope form can help out. Starting with the point-slope equation:

```
(y - y0) = m*(x - x0)
```

multiplying through by m,

```
y - y0 = m*x - m*x0
```

Collecting x and y on the LHS,

```
-m*x + y = y0 - m*x0
```

Moving the constant terms on the RHS to the LHS:

```
(-m)*x + (1)*y + (-1)*y0 + (m)*x0 = 0
```


MATH

The (-1) and the associated multiplications aren't necessary; they just make the extraction of (a,b,c) easier to see.

So it looks like a = -m, b = 1, and c = (m*x0 – y0). In general, any one of the forms may be appropriate for any particular application. However, if you have a problem where you want to compute the intersection of two lines, the general form is usually the most appropriate because you can push them into a matrix and then solve as follows:

```
a1*x + b1*y = c1
a2*x + b2*y = c2
A =   ¦a1  b1¦
      ¦a2  b2¦

X =  ¦x¦
     ¦y¦

B =   ¦c1¦
      ¦c2¦
```

We want to solve **A*X** = **B**. We can use the technique of multiplying each side by the inverse of **A**, $\mathbf{A}^{-1}$.

$$\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

We have done this exercise, so I leave it to you to find a closed form solution.

Playrix Ex. 1023, page 324

07 0672318350 CH04 05•02•2003 2:36 PM Page 300

MATH

If the determinate of **A**, Det(**A**), is zero, then no unique solution exists.

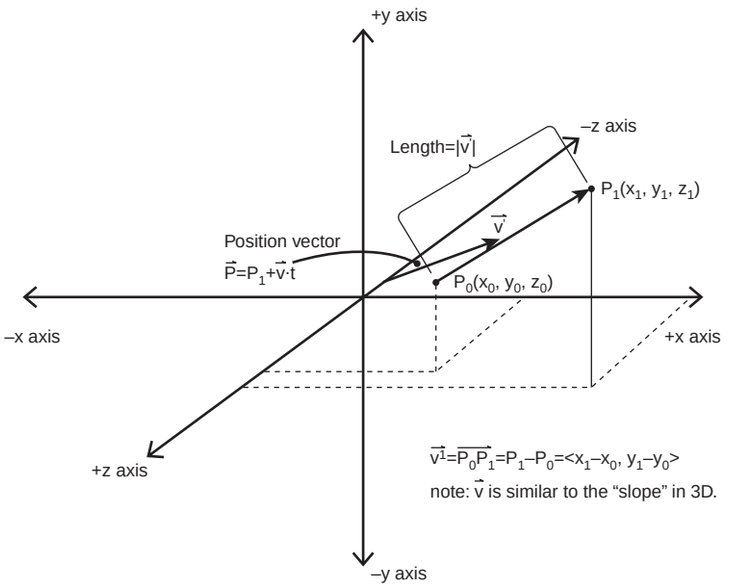


FIGURE 4.31 A line in 3D space.

Lines in 3D Space

Lines in 3D space are a little more complex to represent. In fact, they are downright ugly! In reality, we will use parametric representations of lines in many cases, but I want to talk about parametric representations as a whole; so for now, let's see whether we can figure out a way to represent a 3D line. Take a look at Figure 4.31, which depicts a line in 3D. Referring to the figure, we see a line that passes through points **p0**(x0,y0,z0) and **p1**(x1,y1,z1) and the unit vector **v**' = <x1-x0, y1-y0, z1-z0> = <a,b,c> in the direction of p0 to p1. So we know two points on the line, and we have **v**', which is analogous to the "slope" of the line.

Now, before showing you the equation of the line, let's take a quick look at the vector parametric form just for kicks:

EQUATION 4.22 Parametric Vector Form of a 3D Line

$$\mathbf{p}(x,y,z) = \mathbf{p0} + \mathbf{v}'*t$$

Playrix Ex. 1023, page 325

07 0672318350 CH04 05•02•2003 2:36 PM Page 301

As t varies from zero to |**v**|, **p** traces out **p0** -> **p1**. This should seem familiar, because we covered it when discussing position and 2D displacement vectors a few sections back. Anyway, we are looking for the explicit form of a line in 3D space, and here it is:

EQUATION 4.23 Explicit Symmetrical Form of a 3D Line

```
(x-x0)   (y-y0)   (z-z0)
------ = ------ = ------
   a        b        c
```

Interesting, huh? Basically, there are three equations here, but you only need two because they are coupled. Also, a,b,c are the (x,y,z) components of the unit vector **v**'. The problem with this form is that it's a bit clunky. For example, let's compute it for the line from **p0**(1,2,3) to **p1**(5,6,7).

Given

```
p0(1,2,3)
p1(5,6,7)
v = <p1-p0> = <5-1, 6-2, 7-3> = <4,4,4>
v' = v/¦v¦  = <4,4,4>/¦sqrt(4²+4²+4²)¦
            = <.57, .57, .57>
            = <a,  b,  c>
```

Plugging into Equation 4.23, we have

```
(x-1)     (y-2)   (z-3)
------ = ------ = ------
 .57       .57     .57
```

Multiplying by .57 to clear the denominator and separating the equations, we have

```
(x-1) = (y-2)
```

and

```
(y-2) = (z-3)
```

Simplifying, we get

```
x - y = -1
y - z = -1
```

As you can see in Figure 4.32, lines are fairly ugly in 3D in explicit form. The parametric form of lines is much cleaner.

Playrix Ex. 1023, page 326

07 0672318350 CH04 05•02•2003 2:36 PM Page 302

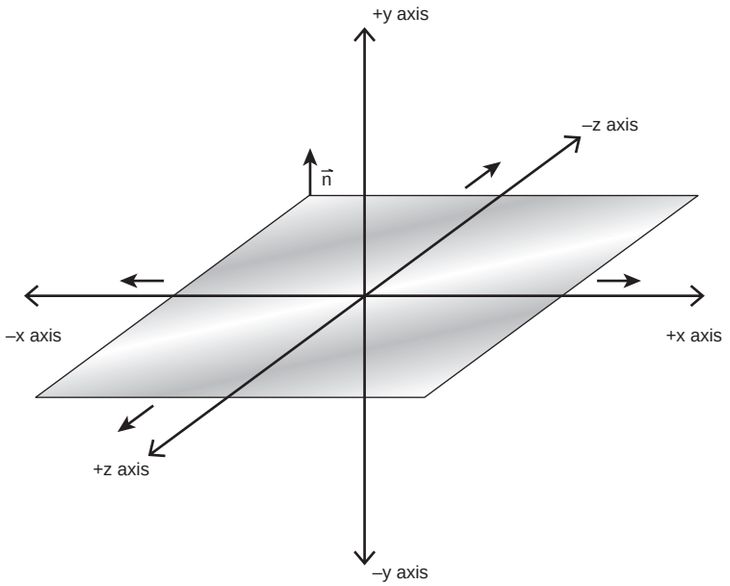


FIGURE 4.32 An infinite plane.

Planes

Planes are really the workhorse of 3D graphics. Technically, a plane is an infinite sheet that extends in all directions infinitely, as shown in Figure 4.32. However, when dealing with 3D graphics, we use the concept of a polygon many times, which is of course a closed (usually coplanar) object that lies in a plane, but is constructed of a set of edges or lines that make it up, as shown in Figure 4.33. However, because all polygons usually lie in a specific plane, we can tell a lot about a polygon from the plane information. Conversely, we can derive the information about a plane in which a polygon lies and use it for mathematical operations as well. Hence, understanding planes is very important. So let's first jot down a couple of the general conceptual attributes of planes:

- All planes are infinite sheets in 3D.
- All planes divide space into two half-spaces (very important for various space partitioning algorithms and collision detection).

A number of methods exist to generate an equation for a plane, but let's take a look at the problem geometrically and see whether we can come up with our own. This is a very important exercise—being able to derive something yourself. It will help you immensely in the future. Anyway, take a look at Figure 4.33, which depicts a plane that has the normal **n** = <a,b,c> with the points **p0**(x0, y0, z0) and **p**(x,y,z) that lie in the plane itself.

Playrix Ex. 1023, page 327

07 0672318350 CH04 05•02•2003 2:36 PM Page 303

We can see the plane we want to define, but what's the equation? Let's begin by noticing that the vector **p0**->**p** lies within the plane, and should (must) be perpendicular (orthogonal) to **n** because both **p0** and **p** lie in the plane. Okay, now we have something to work with—a constraint that we can exploit. We know that for any **p**(x,y,z) in the plane, the vector <**p0**->**p**> dotted with **n** must be equal to zero, because the dot product of two orthogonal vectors is always zero. Therefore, we can write

```
n . (p0->p) = 0
```

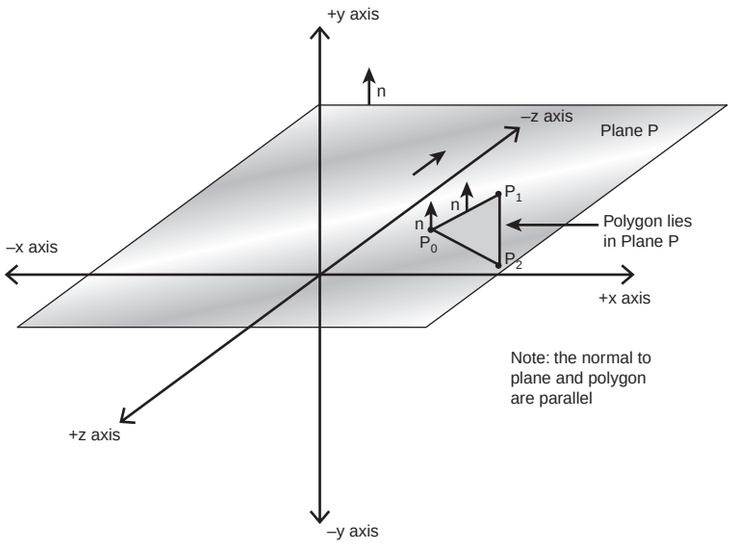


FIGURE 4.33 A plane constructed around a polygon.

That *is* the equation for the plane! Let's multiply it out now and see what it looks like:

EQUATION 4.24 Point-Normal Form of a Plane

```
<a,b,c> . <x-x0, y-y0, z-z0) = 0

a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
```

This is called the *point normal form* of a plane. The reason is that (x0, y0, z0) is a point in the plane, and <a,b,c> is the normal to the plane. Now, if we multiply the terms out and collect the constants, we get the *general form* of a plane:

Playrix Ex. 1023, page 328

07 0672318350 CH04 05•02•2003 2:36 PM Page 304

EQUATION 4.25 General Form of a Plane

Given

```
a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
a*x + b*y + c*z + (-a*x0 - b*y0 - c*z0) = 0
```

and letting d = (-a*x0 - b*y0 - c*z0), we have

```
a*x + b*y + c*z + d = 0
```

The general form comes in handy when you want to compute the intersection of two or more planes. In that case, you have a system of two or more equations in terms of (x,y,z) and you want to solve them. Of course, this is most easily done with the general form.

Computing the Half-Space In Which a Point Is Contained

There is a very important operation that you will use over and over in relation to planes: computing whether a point is on one half-space or the other of a plane. For example, you might want to know whether a projectile has pierced a plane, or something similar. The plane equation can tell you this. Referring to Figure 4.34, we see a plane on its side, along with a point **p** that we want to decide whether it is on the positive half-space or the negative half-space of the plane. The positive half-space is the space into which the normal is pointing, and the negative is of course the other half-space.

To test for this, assuming that we have the equation of the plane in the form:

```
hs = a*(x-x0) + b*(y-y0) + c*(z-z0)
```

All you need to do is plug in the test point for (x,y,z) and test the result. Here are the outcomes:

- If hs = 0, then the point lies in the plane.
- If (hs > 0), then the point lies in the positive half-space.
- If (hs < 0), then the point lies in the negative half-space.

This is one way of writing a polygon point containment test. As an example, let's try a couple of test points, with the x-z plane as the test plane. The normal is <0,1,0> and a point on the plane is (0,0,0):

```
hs = 0*(x-0) + 1*(y-0) + 0*(z-0) = (y-0) = hs
```

First we see that the x-z coordinates are irrelevant, as they should be, because the plane in question is the x-z plane. Therefore, we really only have to look at the y-component of any point. For the point **p**(10,10,10), which is definitely in the positive half-space, we see that (y-y0) = (10-0) = 10 > 0, hence in the positive half-space. So in this case, the problem degenerated into a single component test. In general, it won't, but you get the point. That's what I call a singular sense of humor.

Playrix Ex. 1023, page 329

07 0672318350 CH04 05•02•2003 2:36 PM Page 305

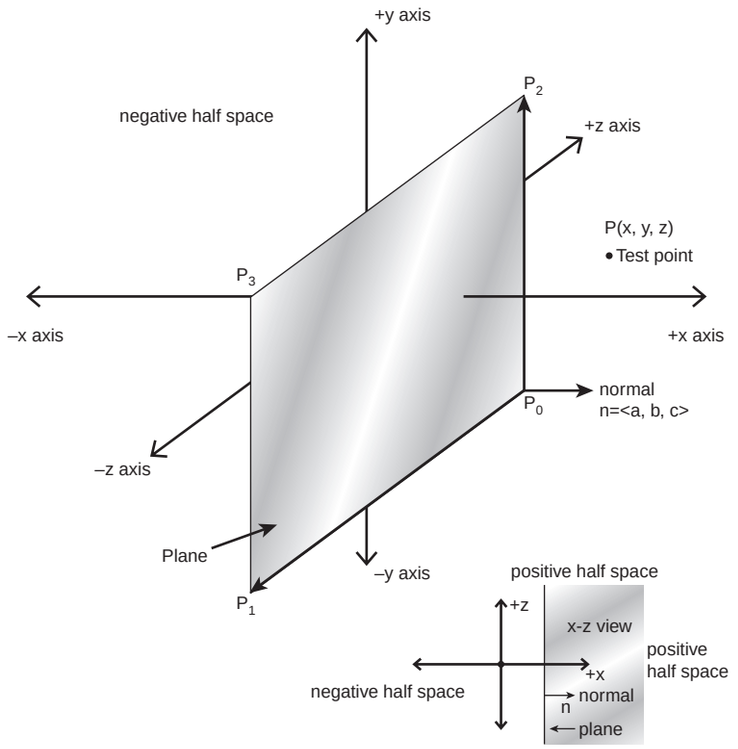


FIGURE 4.34 A plane and its half-spaces.

Intersection of a Plane and Line in 3D with Explicit Representations

As an example, let's see how we would compute the intersection of a line and plane in 3D. This is a very useful operation in a 3D game for a number of reasons: collision detection, clipping, and so forth. We'll use the plane and line shown in Figure 4.35. To make things easy, I have selected the x-z plane as the plane; therefore, the point **p0**(0,0,0) is contained within the plane, and the normal to the plane is **n** = <0,1,0>. The line we want to intersect with the plane goes through the points **p1**(4,4,4) and **p2**(5,-5,-4). Now, let's see whether we can solve for the intersection point $\mathbf{p}_i$(x,y,z) in the figure.

The general point normal equation for the plane is

```
a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
```

Plugging in the normal for <a,b,c> and the point **p0** for (x0, y0, z0), we have

```
0*(x-0)+1*(y-0)+0*(z-0) = 0
                      y = 0
```

Playrix Ex. 1023, page 330

07 0672318350 CH04 05•02•2003 2:36 PM Page 306

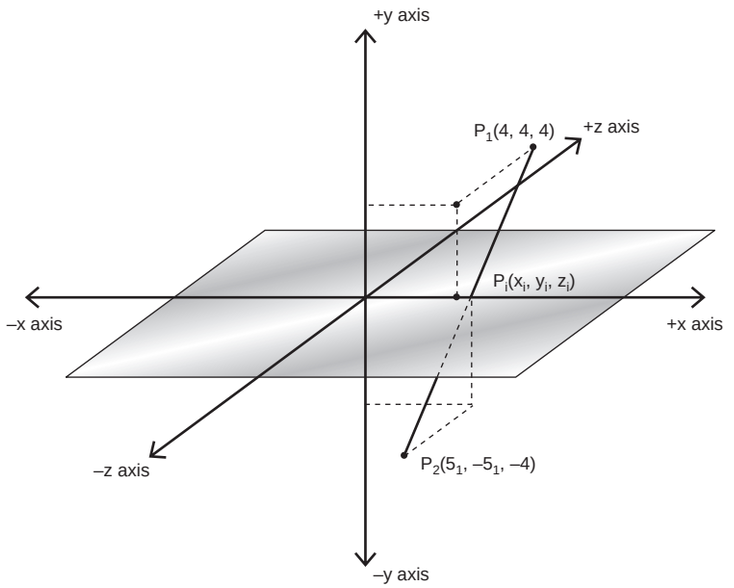


FIGURE 4.35 An illustration of a plane and line intersection.

The general equation for a line in 3D is

```
 (x-x0)  (y-y0)   (z-z0)
------ = ------ = ------
   a        b        c
```

NOTE

The variables are just dummy variables, and represent a point on the line (x0, y0, z0) and a vector parallel to the line <a,b,c>.

The variables <a,b,c> are the unit vector in the direction of the line **p1**->**p2**, which can be computed as follows:

```
v = p2-p1  = (5,-5,-4) - (4,4,4) = <1, -9, -8>
v' = v/¦v¦ = <1, -9, -8>/<sqrt((1)²+(-9)²+(-8)²)
           = <0.08, -0.74, -0.66>
```

Plugging **p1** into (x0,y0,z0) and **v** into (a,b,c), we get

```
(x-4)     (y-4)    (z-4)
------ = ------ = ------
0.08     -0.74    -0.66
```

Playrix Ex. 1023, page 331

07 0672318350 CH04 05•02•2003 2:36 PM Page 307

Collecting both the line equation and plane equation:

1. (x-4)/.08 = -(y-4)/.74 = -(z-4)/.66 (equation of line)
2. y = 0 (plane equation)

We can now solve the system. The plane equation is y = 0, and at this value, the line takes on values:

```
 (x-4)/.08 = -(y-4)/.74
-(y-4)/.74 = -(z-4)/.66
```

At y = 0, we get

```
 (x-4)/.08 = -(0-4)/.74
-(0-4)/.74 = -(z-4)/.66
x = 4+(.08)*4/.74 = 4.43
z = 4 -(0.66)*4/.74 = .43
```


Therefore, the line intersects at $\mathbf{p}_i$(4.43,0,.43), which looks right based on the figure. This example was a bit tedious, but it's important to see it at least once. Later we will derive quicker ways to compute intersections of lines and planes based on *a priori* knowledge of the geometry.

Using Parametric Equations

Parametric equations are infinitely more useful in computer graphics because they can be used to represent lines or curves as a function of a single variable, rather than 2- or 3-space variables. Moreover, they are a more natural method of representing motion and trajectories in computer games. Therefore, let's take a look at them in detail.

2D and 3D Parametric Lines

Referring to Figure 4.36, we see two points in the x-y plane: **p0**(x0, y0) and **p1**(x1, y1). We know that the vector is **v** = <vx, vy> = **p0**->**p1** = (x1-x0, y1-y0). Thus if we were to add **v** to **p0**, we would end up at **p1**. Now, the question is how can we write this parametrically, so that we can vary a value t (for example) in some closed interval [a,b], and so that as we vary the parameter t, a line is traced out from **p0** to **p1** along **v**? Of course, you have already seen this in the vector discussion on position vectors. Once again, the set of points (x,y) from **p0** to **p1** can be described by the following:

EQUATION 4.26 General Parametric Line

```
p = p0 + v*t, for t in the interval [a,b]
```

where **v** = <vx, vy> = **p0**->**p1** = (x1-x0, y1-y0).

Playrix Ex. 1023, page 332

07 0672318350 CH04 05•02•2003 2:36 PM Page 308

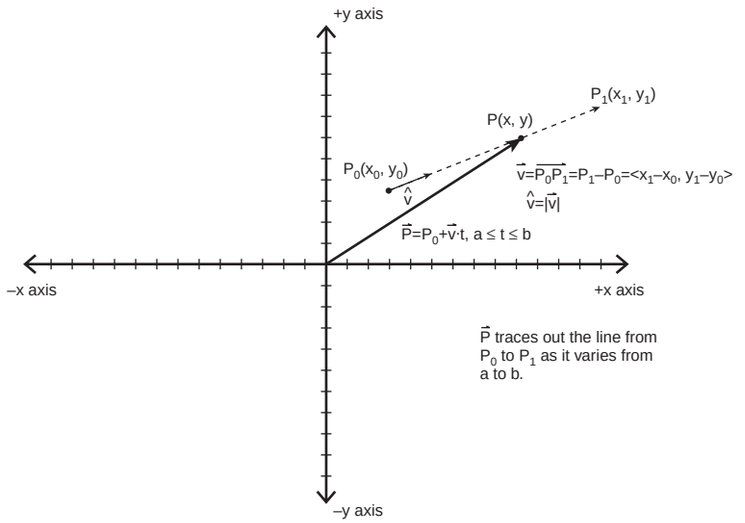


FIGURE 4.36 Parametric line development.

However, the only problem with the formulation is that the interval of *t* is not really known. We can compute it if we want, but there's a very subtle point here. Equation 4.26 is really a general parametric line equation, so lines go from -∞ to +∞; hence, we don't really care about the form of **v**, because we are allowed to set it equal to all real values. However, many times we want more control, with a parametric line segment rather than a parametric line. Now here's the cool part—parametric lines have the exact same form as parametric lines, but we define the interval more precisely, and there are two common ways to adjust the length of **v**. Let's take a look at both.

Parametric Line Segments with Standard Direction Vector v

In the previous discussion, we let **v** = <vx, vy> = **p0**->**p1** = (x1-x0, y1-y0). Therefore, the length of **v** or |**v**| is exactly the length or distance from **p0** to **p1**, as shown in Figure 4.36. So, if we do a little thinking on what values of t define just the line segment from **p0** to **p1**, it's fairly obvious that the interval is [0,1]. Let's plug in the interval and look at the results:

NOTE

The interval notation "[" and "]" means inclusive, and "(" and ")" mean non-inclusive. Thus, [0,1] defines the interval of all real numbers from 0 to 1 including 0 and 1, and the interval (0,1] means all real numbers from 0 to 1 including 1, but excluding 0.

Playrix Ex. 1023, page 333

07 0672318350 CH04 05•02•2003 2:36 PM Page 309

We have the following formula for the parametric line:

p = **p0** + **v***t, for *t* in the interval [0,1]

where

v = <vx, vy> = **p0**->**p1** = (x1-x0, y1-y0)

At t = 0, we get the following result:

p = **p0** + **v***t = **p0**

which is correct.

And at t = 1, we get the following result:

p = **p0**+**v***1 = **p0**+**v** = **p0**+<vx, vy> = **p1**

which is also correct.

Hence, the interval [0,1] exactly traces the line from **p0** to **p1** as *t* varies from 0 to 1. Additionally, we see that it's perfectly legal to plug values into *t* that are beyond the interval, but these values are not on the desired segment. However, note that this is a very important property when trying to solve parametric systems. If you design a line segment interval from [0,1] and you get a parameter *t* outside of the interval for some calculation, then you know that the point is outside of the segment—this is a very important detail.

Also, using the standard v and interval from [0,1] is nice because 0 and 1 are both fantastic numbers! Now, let's see the other representation of a parametric line using a normalized v.

Parametric Line Segments with Unit Direction Vector |v| = 1

The next version of a parametric line is again the same as the previous, but with the one change that we normalize the direction vector **v**, call it **v**' so that |**v**'| = 1. Okay, let's see the equations with the normalized **v**' replacing v:

p = **p0** + **v**'***t**, for *t* in the interval [a,b]

where **v** = <vx, vy> = **p0**->**p1** = (x1-x0, y1-y0) and **v**' = **v**/|**v**|.

Now, here's the problem: What is the interval that defines the line segment from **p0** to **p1**, as shown in Figure 4.37? If you're keeping awake, you should immediately see that the interval is [0, |**v**|]! Let's see why.

When t = 0, we have

p = **p0** + **v**'*0 = **p0**

Playrix Ex. 1023, page 334

07 0672318350 CH04 05•02•2003 2:36 PM Page 310

which is correct. When t = |v|, we have

```
p = p0 + v'*¦v¦ = ?
```

which is equivalent to

```
          v
 = p0 + --- * ¦v¦
        ¦v¦
```

The |**v**|'s cancel out, giving us

```
= p0 + v
```

Plugging in for **v**, we get

```
p = p0+v = p0+<vx, vy> = p1
```

which is what we wanted. Ultimately, there is no difference between the two representations. They are both constructed the same, but in different intervals, so that in one case we vary *t* from [0,1], and in the other, we vary *t* from [0,|**v**|]. You could also construct a parametric so that as you varied *t* from [-1, 1], you would get the line from **p0** to **p1**. Try and see whether you can come up with the formula for this parametric line yourself.

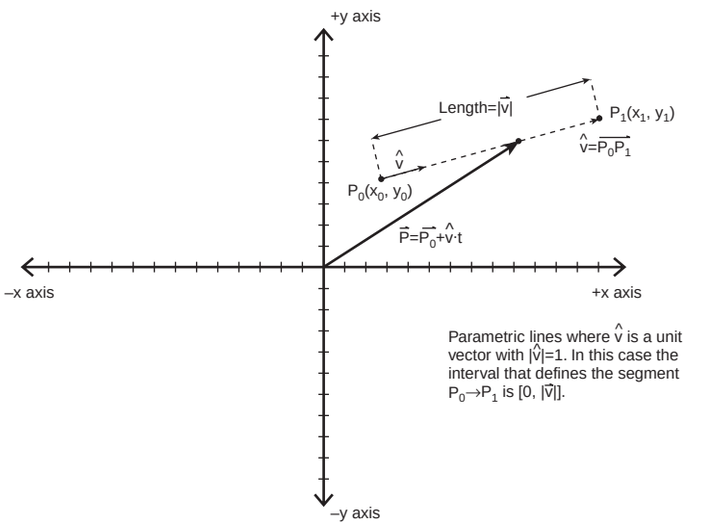


FIGURE 4.37 The interval that defines the line **p0** to **p1.**

Playrix Ex. 1023, page 335

07 0672318350 CH04 05•02•2003 2:36 PM Page 311

Parametric Lines in 3D

3D parametric lines are absolutely identical to the 2D version: The points **p0**(x0, y0, z0) and **p1**(x1, y1, z1) have three components, and of course so does **v** = <vx, vy, vz> = <x1-x0, y1-y0, z1-z0>. The vector equation stays the same:

`p = p0 + v*t`, for *t* in the interval [a,b]

where

`v = <vx, vy, vz> = p0->p1 = (x1-x0, y1-y0, z1-z0)`

Other than the new parameter, everything stays the same. However, notice the simplicity from moving from 2D to 3D. Remember how complex a 3D explicit line was in terms of x, y, and z? It was ugly, but in parametrics, it's very simple. That's why working with lines in parametric form in a 3D engine is almost necessary, because the explicit form is nearly impossible to represent and do calculations on.

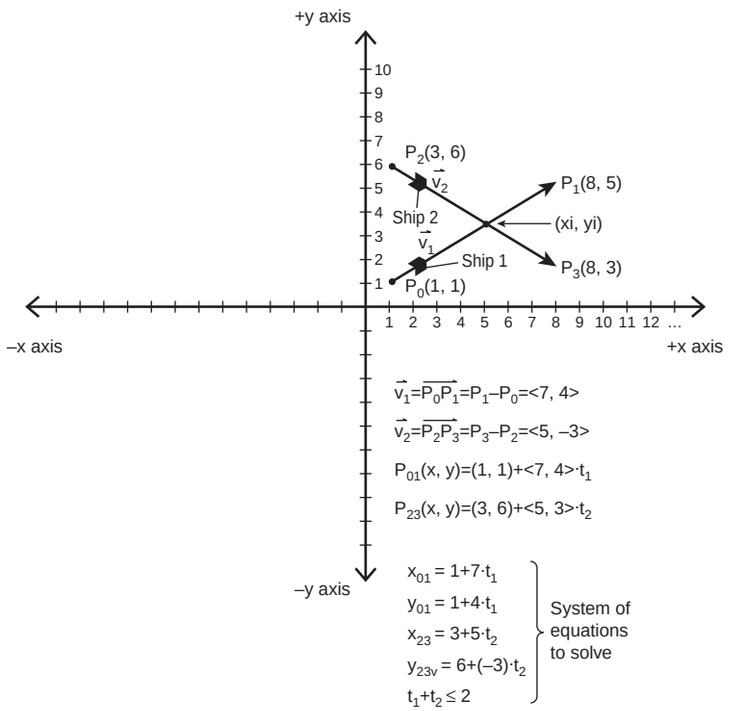


FIGURE 4.38 A pair of ships on 2D trajectories.

Playrix Ex. 1023, page 336

07 0672318350 CH04 05•02•2003 2:36 PM Page 312

Computing the Intersection of Parametric Lines

Now, let's say that you have two ships traveling in 2D, as shown in Figure 4.38. From the figure, we see that:

- Ship 1 is traveling from **p0**(x0,y0) to **p1**(x1,y1).
- Ship 2 is traveling from **p2**(x2,y2) to **p3**(x3,y3).

We want to determine whether their trajectories will intersect. Now, we haven't said anything about whether the ships actually intersect because time hasn't been included into the problem. Remember, the objects not only have to collide in space, but at the same time! For now, let's just see whether their trajectories intersect. Your first attempt might be to construct two explicit lines for each trajectory and then see whether the lines intersect, but then you're hosed, because how can you tell whether the intersection is on the segments from **p0**->**p1** and **p2**->**p3**? That's why parametric lines are so useful. We can perform the calculations and find the solution value *t* at the intersection. However, if *t* is not in the interval that defines the lines, we know that their segments don't intersect. This case is shown in Figure 4.39.

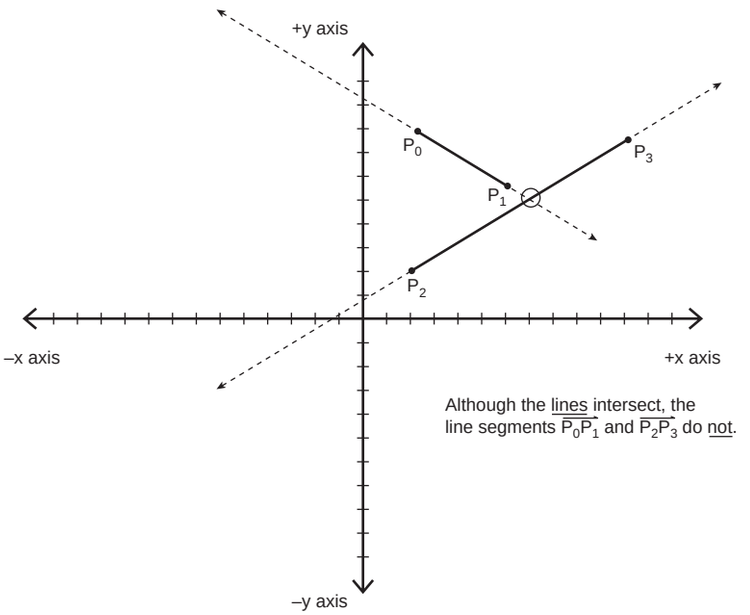


FIGURE 4.39 The point of intersection must lie on the lines' segments.

Let's begin by constructing the two parametric lines with intervals from [0,1] to make it easier. Note that if the lines each had different intervals, we would be comparing apples to oranges mathematically. Here are the definitions of the two lines parametrically:

Playrix Ex. 1023, page 337

07 0672318350 CH04 05•02•2003 2:36 PM Page 313

Given

- Ship 1 is traveling from **p0**(1,1) to **p1**(8,5).
- Ship 2 is traveling from **p2**(3,6) to **p3**(8,3).

Computing the direction vectors for each line:

```
v1 = p0->p1 = <x1-x0, y1-y0> = <8-1, 5-1> = <7,4>
v2 = p2->p3 = <x3-x2, y3-y2> = <8-3, 3-6> = <5,-3>
```

For the parametric line equations, we have

```
p01 = p0 + v1*t1, t1 ∈ [0,1]
p23 = p2 + v2*t2, t2 ∈ [0,1]
```


And with the real values:

```
p01(x,y) = (1,1) + <7,4>*t1
p23(x,y) = (3,6) + <5,-3>*t2
```

Notice that the parameter variable is different in each line. We have *t1* and *t2* rather than *t*. This is because the parameters *are* different. It's true that each parameter varies in the interval from [0,1], but other than that, the *t* value on line one and the *t* value on line two where the intersection occurs (if any) need not be the same. Hence, we must track this by using two different parameters—this is very important, and forgetting to do it will cause lots of math problems.

Let's write all the equations in terms of their components:

```
x = 1 + 7*t1
y = 1 + 4*t1
```

and

```
x = 3 + 5*t2
y = 6 - 3*t2
```

We want to equate the x,y components together for each line and solve for *t1* and *t2*, then see whether *t1* and *t2* are both in the interval [0,1]. If so, we have an intersection, otherwise we don't. Equating terms, we have

```
x: 1 + 7*t1 = 3 + 5*t2
y: 1 + 4*t1 = 6 - 3*t2
```

Rearranging, we get

```
7*t1 - 5*t2 = 2
4*t1 + 3*t2 = 5
```

Playrix Ex. 1023, page 338

07 0672318350 CH04 05•02•2003 2:36 PM Page 314

We have two equations and two unknowns, so we can solve using whatever technique we want: Gaussian elimination, Cramer's Rule, substitution, voodoo magic, whatever. The results are

```
t1 = .756
t2 = .658
```

We see that the lines intersect, because both *t1* and *t2* are in the interval of [0,1]. If we want the actual intersection point, we can plug *t1* or *t2* into either line equation. Let's plug both *t1* and *t2* into each and make sure that both result in the same (x,y), just to be safe:

For line 1:

```
x = 1 + 7*t1 = 1 + 7*(.756) = 6.29
y = 1 + 4*t1 = 1 + 4*(.756) = 4.02
```

And for line 2:

```
x = 3 + 5*t2 = 3 + 5*(.658) = 6.29
y = 6 - 3*t2 = 6 - 3*(.658) = 4.02
```

They check, thank Zeus! That should be enough to convince you that parametric representations are the best, and are simple to work with. Additionally, the same problem in 3D is absolutely identical. There are no weird formulas, just another variable. Hence, in 3D you would end up with something like this for each line:

```
p01(x,y,z) = (x0,y0,z0) + <vx1,vy1,vz1>*t1
p23(x,y,z) = (x2,y2,z2) + <vx2,vy2,vz2>*t2
```

And then you would break these into three equations and solve for *t1* and *t2*:

For line 1:

```
x = x0 + vx1*t1
y = y0 + vy1*t1
z = z0 + vz1*t1
```

For line 2:

```
x = x2 + vx2*t2
y = y2 + vy2*t2
z = z2 + vz2*t2
```

The only difference is that you have three equations and two unknowns, so you can pick any pair of the variables x,y,z and then solve for *t1* and *t2*. After you have solved the system, you must make sure that *t1* and *t2* result in the same point when plugged into the

Playrix Ex. 1023, page 339

07 0672318350 CH04 05•02•2003 2:36 PM Page 315

third equation variable that you didn't use. However, in most cases, you will never need to detect whether a 3D line intersects another 3D line segment because the probability of this happening in any context is remote. Most of the time, you will want to detect whether a 3D parametric line intersects with a plane. Let's take a look at this problem.

Computing the Intersection of a 3D Parametric Line and a 3D Plane

Computing the intersection of a 3D line and a plane can be extremely simple or fairly complex, depending on the plane. If the plane is x-y, x-z, y-z, or a plane parallel to them, it's trivial. However, for a general line plane intersection, we are going to have to work harder.

Let's start by writing down the formula for a parametric line and plane and solve the trivial case first. Here's the parametric line equation in 3D:

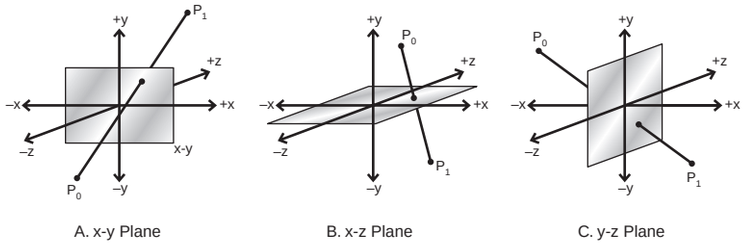
```
p = p0 + v*t, t ∈ [0,1]
```


Or more explicitly:

```
p(x,y,z) = p0(x0,y0,z0) + v<vx, vy, vz>*t
```

And here's the 3D plane equation:

```
n . (p - p0) = 0
nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0
```



A. x-y Plane B. x-z Plane C. y-z Plane

FIGURE 4.40 Intersections of lines and planes (trivial cases).

Referring to Figure 4.40, we see the three cases we are interested in:

- Case 1: Intersection of the line and the x-y plane when z = 0.
- Case 2: Intersection of the line and the x-z plane when y = 0.
- Case 3: Intersection of the line and the y-z plane when x = 0.

Solving all of these is trivial. Let's solve the x-y plane case as an example. In this case, we know that the z-coordinate must be 0, thus we can plug this into the parametric line equation for the z-component and see what *t* we get:

```
p(x,y,z) = p0(x0,y0,z0) + v<vx, vy, vz>*t
```

Playrix Ex. 1023, page 340

07 0672318350 CH04 05•02•2003 2:36 PM Page 316

Component-wise:

```
x = x0 + vx*t
y = y0 + vy*t
z = z0 + vz*t
```

Plugging z = 0 into the z equation, we get

```
z = z0 + vz*t = 0
```

or

```
t = -z0/vz
```

If *t* is in the interval [0,1], then we have an intersection and we can plug the *t* into the x and y components of the parametric line and find those values, too:

```
x = x0 + vx*(-z0/vz)
y = y0 + vy*(-z0/vz)
```

Thus, intersecting lines with planes that are parallel to any of the standard planes in 3D is trivial—the problem is intersecting a 3D line and an arbitrary plane. Hmmm, let's take a look at the line again:

```
p(x,y,z) = p0(x0,y0,z0) + v<vx, vy, vz>*t
```

Component-wise:

```
x = x0 + vx*t
y = y0 + vy*t
z = z0 + vz*t
```

And here's an arbitrary plane:

```
                          n . (p - p0) = 0
        nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0

nx*x + ny*y + nz*z + (-nx*x0-ny*y0-nz*z0) = 0
```

Written in general form:

```
a*x + b*y +c*z + d = 0
```

where a = nx, b = ny, c = nz, and d = (-nx*x0-ny*y0-nz*z0).

CAUTION

Don't confuse the x0, y0, z0 of the plane and the line—they are different! That's why I keep d in symbolic form, rather than long form.

Playrix Ex. 1023, page 341

07 0672318350 CH04 05•02•2003 2:36 PM Page 317

Is the solution jumping out at you yet? If we plug in the values of the right-hand sides of x,y,z from the parametric line into the general form of the plane for x,y,z, we can solve for *t*! Here's the math:

EQUATION 4.27 Intersection of a Parametric Line and a Plane

```
a*(x0+vx*t) + b*(y0+vy*t) + c*(z0+vz*t) + d = 0
```

Solving for *t*:

```
t = -(a*x0 + b*y0 + c*z0 + d)/(a*vx + b*vy + c*vz)
```

where (x0,y0,z0, vx,vy,vz) are from the parametric line, and (a,b,c,d) are from the plane.

And of course, you just plug that back into the parametric equations to find the actual (x,y,z):

```
x = x0 + vx*t
y = y0 + vy*t
z = z0 + vz*t
```

The cool thing is that this *t* will be the solution no matter what interval the *t* of the segment is defined as. Thus, after you have *t*, just test whether it's in your interval [0,1] or whatever. The math will scale *t* to your interval no matter what, because the solution for *t* has the direction vector in it. Of course, watch out for a zero denominator before the division!

NOTE

Although I came up with the solution algebraically, it's more than obvious that a more beautiful geometric/vector solution is possible, because it's not hard to see that both the numerator and denominator are dot products—can you find one? If so, email it to me!

At this point, you are armed with enough math to handle 3D collision detection, wall following, clipping, weapon projectiles, and a lot more. We are going to use this stuff a lot later in the book, so make sure you understand it forward and backward.

Introduction to Quaternions

I remember talking to another game programmer in the early 80s about the mysterious speed of Atari's *Space Duel*. I couldn't believe how fast the 3D was working. My friend had some inside information that Atari was using "Matrix Multiplying." I was pretty impressed by that, because doing real math on 8-bit processors was quite a feat. These days, we have BSP trees, portals, and A-buffers, but you can't go anywhere without hearing the word *quaternion*.

Playrix Ex. 1023, page 342

07 0672318350 CH04 05•02•2003 2:36 PM Page 318

Quaternions were invented in the 1800s by the mathematician William Rowan Hamilton (if you study graph theory, you should be familiar with Hamiltonian paths). Quaternions weren't specially designed for 3D graphics (obviously), but they have found their application in a couple of areas of 3D graphics for which they are well suited:

- 3D camera control
- Compact storage
- Smooth 3D interpolation

Quaternions are based on complex numbers, which are themselves a bit abstract to understand. And of course, they don't really mean anything in the real world—they only exist mathematically. However, complex numbers are a tool that can be used to represent mathematical ideas for which real numbers just don't work. With that in mind, first we are going to take a look at basic complex number theory, and then move on to quaternions and their mathematical properties. After that we'll see how they are applied in computer graphics and games.

Complex Number Theory

The set of real numbers $\boldsymbol{R}$ consists of all numbers in the interval [-∞, +∞]. Easy enough, but take a look at these equations:

```
x = sqrt(4) = 2
x = sqrt(1) = 1
x = sqrt(-1) = ???
```

The third equation is the problem—we don't have a number in the set of real numbers for the square root of –1 because there is no number that when multiplied by itself is equal to –1. You might say –1, but –1 * –1 = 1, so that's not correct. We need a new number, which we will call the *imaginary number i* (or *j* for electrical engineers). We are going to create the following rule:

```
let i = sqrt(-1)
```

And then

```
i*i = -1
```

Now we can compute things like sqrt(-4), which would be

```
sqrt(-4) = 2*i
```

because

$(2*i)*(2*i) = 4*i^2 = 4*-1 = -4$

Playrix Ex. 1023, page 343

07 0672318350 CH04 05•02•2003 2:36 PM Page 319

Now, the cool thing about the imaginary number *i* is that you can think of it as a variable or coefficient; that is, all the rules of normal algebra still hold. You just pretend *i* is a variable like *x* and do what you would normally do. Then whenever you see i^2 terms, you can convert them into –1 if you like to simplify. For example, look at the following sum:

```
3 + 5*i + 3*i² – 10 + 6*i
```

Collecting terms, we get

```
= 3*i² + 5*i + 6*i + 3 – 10
 = 3*i² + 11*i - 7
 = 3*(-1) + 11*i – 7
 = -3 + 11*i – 7
 = -10 + 11*i
```


Nothing unordinary there. But, the imaginary number by itself is pretty boring, so mathematicians came up with the concept of a *complex number*, which is the sum of a real number and an imaginary one. Mathematically, a complex number looks like this:

```
z = (a + b*i)
```

a is called the *real part (RP)*, and *b* is called the *imaginary part (IM)*. Because *a* and *b* can never be added together because of the imaginary coefficient of *b*, you can also think of complex numbers as points in the complex plane, as shown in Figure 4.41.

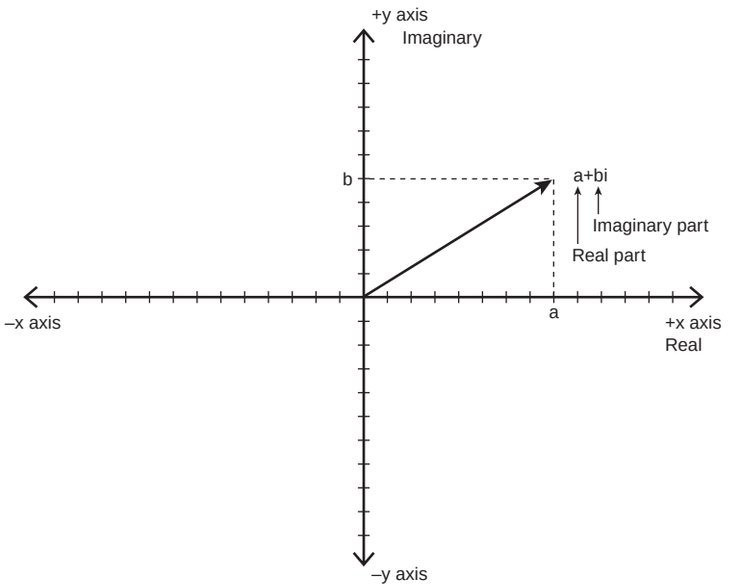


FIGURE 4.41 The complex plane.

Playrix Ex. 1023, page 344

07 0672318350 CH04 05•02•2003 2:36 PM Page 320

As a convention, most people call the x-axis the real part, and the y-axis the imaginary axis. Thus, we have a geometrical interpretation of complex numbers based on the vector basis:

```
z = a*(1,0) + b*(0,i)
```

I want to come back to this concept a little later, so keep that in mind. However, let's look at the operations on complex numbers and figure out how to add, subtract, multiply, divide, and so on.

Complex Scalar Multiplication/Division

Multiplication of a scalar and a complex number is performed on a component-wise basis as follows:

Given:

$\mathbf{z}_1$ = (a+b*i)
k*$\mathbf{z}_1$ = k*(a+b*i) = (k*a + (k*b)*i)

Example:

```
3*(2+5*i) = (6 + 15*i)
```

That is, the scalar simply distributes over the complex number. Division of course is similar, because you can think of it as multiplication by the inverse.

Complex Addition and Subtraction

To add or subtract complex numbers, you simply add or subtract the real and the imaginary parts separately, as follows:

Given:

$\mathbf{z}_1$ = (a+b*i)
$\mathbf{z}_2$ = (c+d*i)
$\mathbf{z}_1$ + $\mathbf{z}_2$ = (a+b*i) + (c+d*i) = ((a+c) + (b+d)*i)

Example:

```
(3 + 5*i) + (12 - 2*i) = (15 + 3*i)
```

Complex Additive Identity

Also, the *additive identity* (the complex number that, when added to any other complex number, is that number) is (0+0*i), because:

```
(0+0*i) + (a+b*i) = (0+a + (0+b)*i) = (a+b*i)
```

Playrix Ex. 1023, page 345

07 0672318350 CH04 05•02•2003 2:36 PM Page 321

Complex Additive Inverse

The *complex additive inverse* (the number that, when added to any other complex number, is the additive identity (0 + 0*i)) is $\mathbf{z}^*$ = (-a - b*i) for any complex number **z** = (a + b*i) because:

```
(a + b*i) + (-a – b*i) = (a-a) + (b-b)*i) = (0 + 0*i)
```

Complex Multiplication

Now for the fun part! Multiplying complex numbers is actually very easy. Let's do it, and pretend like *i* is a variable, as we have before.

$\mathbf{z}_1$ = (a+b*i)
$\mathbf{z}_2$ = (c+d*i)

$\mathbf{z}_1 * \mathbf{z}_2$ = (a+b*i) * (c+d*i)
= (a*c + a*d*i + b*c*i + b*d*i^2)

Noting that $i^2 = -1$ and collecting terms, we get

```
= (a*c + (a*d+b*c)*i + b*d*(-1))
```

Collecting terms once again, we have

```
= ((a*c-b*d) + (a*d+b*c)*i)
```

Therefore, we see that the real part equals (a*c - b*d), and the imaginary part equals (a*d + b*c). Let's do an example:

Example:

```
(1+2*i) * (-3 + 3*i) = ((-3-6) + (-6+3)*i) = (-9 - 3*i)
```

Complex Division

Division of complex numbers can be accomplished using brute force. For example, we can compute the quotient of two complex numbers $\mathbf{z}_1/\mathbf{z}_2$ in the following way:

$\mathbf{z}_1$ = (a+b*i)
$\mathbf{z}_2$ = (c+d*i)

$$\mathbf{z}_1/\mathbf{z}_2 = \frac{(a+b*i)}{(c+d*i)}$$

If c = 0 or d = 0, then the division is trivial, but if neither c nor d is equal to 0, then we are at a bit of an impasse. The question is how to perform the division so that the result is in the form (a + b*i) once again. The trick is to clear the denominator first and turn it into a

Playrix Ex. 1023, page 346

07 0672318350 CH04 05•02•2003 2:36 PM Page 322

pure scalar. This way, we can simply divide the scalar into the real part and imaginary part of the numerator. To turn the denominator into a pure scalar, we must multiply by the *complex conjugate*, usually denoted by a superscript asterisk (*).

The complex conjugate of $\mathbf{z} = (a + b*i)$ is $\mathbf{z}^* = (a - b*i)$. When we multiply a complex number and its conjugate, the result is always a pure real number. Take a look:

Given:

$$\mathbf{z} = (a+b*i)$$

then

$$\mathbf{z} * \mathbf{z}^* =$$
$$(a+b*i) * (a-b*i) = (a^2 + a*b*i - a*b*i - b^2*i^2)$$
$$= (a^2 + b^2)$$

Cool, huh? With this trick we can convert the quotient problem into a more palatable form:

Given the quotient of two complex numbers

$$\frac{(a+b*i)}{(c+b*i)}$$

We can always multiply by 1 and not change the results, so let's multiply by the complex conjugate of the denominator over itself:

$$\frac{(a+b*i)}{(c+d*i)} * \frac{(c-d*i)}{(c-d*i)}$$

Lots of steps go here...

$$= \left(\frac{(a*c+b*d)}{(a^2 + b^2)} + \frac{((b*c-a*d)*i)}{(a^2 + b^2)} \right)$$

Ugly as it might seem, rest assured it's of the form (a+b*i).

Multiplicative Inverse

The last mathematical property—complex numbers need to be a closed set—is a *multiplicative inverse*, which is a complex number that, when multiplied by another, results in "1"—where "1" in complex numbers is equal to (1+0*i). If you think about it long enough, you

Playrix Ex. 1023, page 347

07 0672318350 CH04 05•02•2003 2:36 PM Page 323

will definitely see that we can use the complex conjugate in some way. Let's just try the obvious and write an equation and see what happens. We want the following:

```
(a+b*i) * (c+d*i) = (1+0*i)
```

Amazingly, the multiplicative inverse of **z** = (a+b*i) is just

```
        1
1/z = ------
      (a+b*i)
```

But the problem is that we no longer have something in the form (real_part + imaginary_part*i); we have its inverse. We need to reform the preceding function so that it's of the form (a + b*i). We can multiply by the complex conjugate to find them, resulting in

```
   1    (a-b*i)        a          b
------  ------- = -------- + -------- * i
(a+b*i) (a-b*i)   (a² + b²)   (a² + b²)
```

Gnarly, huh? What this means is that if you are given the complex number (a+b*i), its inverse is 1/(a+b*i), but if you need the real part and imaginary part, you need to use the preceding equation.

Complex Numbers as Vectors

Last but not least, I want to revisit the representation of complex numbers and show you another way of thinking of them (if you haven't already figured it out). Complex numbers can also be thought of as vectors in a 2D plane. If you take a look at Figure 4.41, we've plotted an imaginary number on a 2D Cartesian plane with the real part as the x-coordinate and the imaginary part as the y-coordinate. So, there's no reason why we can't think of imaginary numbers as vectors with the basis:

```
z = a*(1,0) + b*(0,i)
```

Or, more compactly:

```
z = <a,b>
```

where *a* is the real part, and *b* is the imaginary part. Figure 4.42 shows our new representation in vector form.

The cool thing about representing complex numbers as vectors is that we can then transform them as vectors, and the results will all be perfectly valid. Moreover, representing complex numbers as vectors allows us to better visualize an abstract mathematical notion such as complex numbers, and to thus "see" relationships that might otherwise elude us in their pure mathematical form.

Playrix Ex. 1023, page 348

07 0672318350 CH04 05•02•2003 2:36 PM Page 324

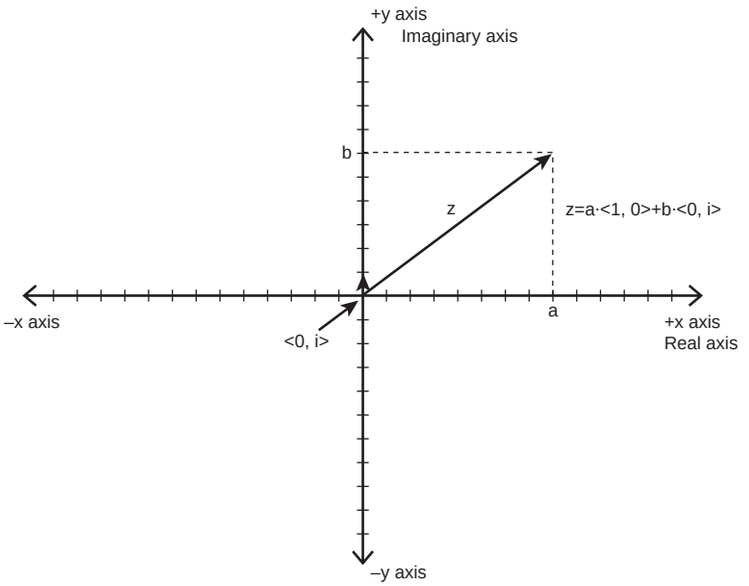


FIGURE 4.42 Complex numbers in a unit vector form.

Norm of a Complex Number

Many times we want to know the length of a complex number or the magnitude. Of course, this doesn't make much sense in a purely mathematical sense, but when we think of complex numbers as vectors in space, it's very natural and easy to understand. Anyway, the *norm* of a complex number can be computed in the following ways:

EQUATION 4.28 Norm of a Complex Number

Given

$\mathbf{z} = a+b*i$

then

$|\mathbf{z}| = \text{sqrt}(a^2+b^2)$

Also, take a look at the product of any complex number and its conjugate:

Given

$\mathbf{z} = a+b*i,\ z^* = a-b*i$

then

$\mathbf{z} * \mathbf{z}^* = a^2 + b*i - b*i + b^2 = a^2+b^2$

Playrix Ex. 1023, page 349

07 0672318350 CH04 05•02•2003 2:36 PM Page 325

Therefore, the norm is also equal to

$|\mathbf{z}| = \text{sqrt}(\mathbf{z} * \mathbf{z}^*)$

Hyper-Complex Numbers

Quaternions are nothing more than *hyper-complex* numbers. The term *hyper-complex* can really mean anything mathematically, but usually it means a complex number that has more than one imaginary component. In our case, we are going to refer to hyper-complex numbers with one real part and three imaginary parts—otherwise known as quaternions.

A quaternion can be written many ways, but in general it's of the form seen in Equation 4.29:

EQUATION 4.29 The Form of a Quaternion

$$\mathbf{q} = q_0 + q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$$

or

$$\mathbf{q} = q_0 + \mathbf{q}_v, \text{ where } \mathbf{q}_v = q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$$

i = <1,0,0>, **j** = <0,1,0>, **k** = <0,0,1>, and $<q_0,q_1,q_2,q_3>$ are all real numbers, and **i**, **j**, and **k** are all imaginary numbers that form the vector basis of the quaternion **q**. Moreover, q_0 is real, and has no imaginary coefficient.

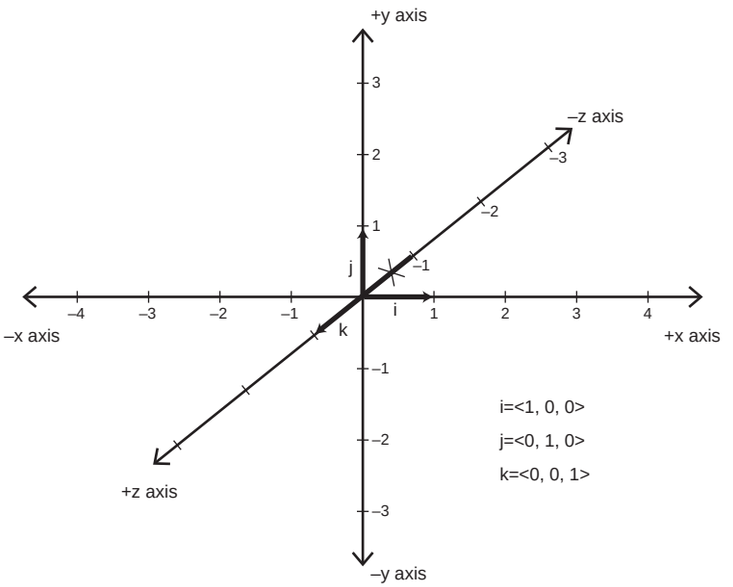


FIGURE 4.43 The hyper-complex basis interpreted as a 3D system.

Playrix Ex. 1023, page 350

07 0672318350 CH04 05•02•2003 2:36 PM Page 326

Also, the imaginary basis <**i**,**j**,**k**> has some interesting properties. It can be thought of as a 3D mutually perpendicular set of unit vectors in an imaginary coordinate system that locates points in <**i**,**j**,**k**> space, as shown in Figure 4.43. But, the interesting thing about <**i**,**j**,**k**> is the following relationship:

EQUATION 4.30 The Quaternion Basis Products

$$i^2 = j^2 = k^2 = -1 = i*j*k$$

MATH

Note that I am bolding the imaginary basis **i**, **j**, **k**; thus, we would think they are vectors, and they are, but they can also be thought of as variables. They have a dualistic nature, and sometimes it's appropriate to think of them in their vector form, and sometimes it's not. I might switch back and forth, especially when I am using the i, j, or k alone or showing relationships among them.

The " = **i*j*k**" part takes a little getting used to, but it's correct. And of course, you can manipulate the relationship to arrive at the following:

Given

$$i^2 = j^2 = k^2 = -1 = i*j*k$$

therefore

$$i = j*k = -k*j$$
$$j = k*i = -i*k$$
$$k = i*j = -j*i$$

which intuitively seems right, because the cross-product of any two perpendicular vectors in an orthonormal basis should be perpendicular to the vectors themselves, or the third vector in a 3D orthonormal basis. And if we invert the order of multiplication, we should see an inversion or sign flip, and we do.

At this point, I want to establish some conventions about writing quaternions—otherwise, we are going to get caught up in notation. Many people like to write quaternions using lowercase letters to represent them, but they also break them up into a real and imaginary part, where they represent the imaginary part as a vector like this:

$$\mathbf{q} = q_0 + \mathbf{q}_v$$

where

$$\mathbf{q}_v = q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$$

Playrix Ex. 1023, page 351

07 0672318350 CH04 05•02•2003 2:36 PM Page 327

So q_0 is the real part, and $\mathbf{q}_v$ is the imaginary part and is a vector $<q_1, q_2, q_3>$. That's cool, but I don't like the fact that **q**, the quaternion itself, is not bolded—it's confusing because it's a 4-tuple itself. We are going to use a notation in which quaternions are represented in lowercase bold, AND the vector part is also in lowercase bold. Here's an example:

```
a = -1 + 3*i + 4*j + 5*k
```

which in pure vector form is

```
a = <-1, 3, 4, 5>
```

or in real-vector form is

$\mathbf{a} = q_0 + \mathbf{q}_v$

where $q_0 = -1$, and $\mathbf{q}_v = <3,4,5> = 3*i + 4*j + 5*j$.

The point is that we are going to use arrays obviously to represent quaternions in real life, and the first number will always be the real part, and the remaining three numbers will always be the imaginary coefficients of <i,j,k>. Moreover, from now on I want to start getting you into the habit of representing quaternions in this format:

$\mathbf{q} = q_0 + x*\mathbf{i} + y*\mathbf{j} + z*\mathbf{k}$

Or in another way:

$\mathbf{q} = q_0 + <x,y,z> . <\mathbf{i},\mathbf{j},\mathbf{k}>$

NOTE

Note that here we are taking advantage of the vector property of <**i**,**j**,**k**>.

That is, simply referring to the real part as q0 and the imaginary part as $\mathbf{q}_v = <x,y,z>$ gives us a little more to hold on to when relating quaternions to 3D space. However, depending on what we are doing, we might flip from one representation to another, but I think you get the point.

The cool thing about quaternions (or any hyper-complex number system) is that the mathematics of addition, multiplication, inverses, and so on are all the same as in standard complex number theory, but with more elements. Hence, we really already know how to do this stuff, but we just have to take into consideration that we now have three imaginary components rather than one, as we did with basic complex numbers or complex numbers of "rank 1."

Now let's take a look at the basic operations that we might need to perform on quaternions when doing calculations—I'm going to go fast because this should all be obvious at this point.

Playrix Ex. 1023, page 352

07 0672318350 CH04 05•02•2003 2:36 PM Page 328

Quaternion Addition and Subtraction

Addition or subtraction of quaternions is accomplished by adding or subtracting the real part and the imaginary part, just as with normal complex numbers.

Example:

$\mathbf{q} = q_0 + \mathbf{q}_v$
$\mathbf{p} = p_0 + \mathbf{p}_v$

$\mathbf{q} + \mathbf{p} = (q_0+p_0) + (\mathbf{q}_v+\mathbf{p}_v)$

Example:

$\mathbf{q} = 3 + 4*\mathbf{i} + 5*\mathbf{j} + 6*\mathbf{k}$ = <3,4,5,6> in vector form.
$\mathbf{p} = -5 + 2*\mathbf{i} + 2*\mathbf{j} - 3*\mathbf{k}$ = <-5,2,2,-3> in vector form.

$\mathbf{q} + \mathbf{p} = (3+ -5) + ((4+2)*\mathbf{i} + (5+2)*\mathbf{j} + (6+ -3)*\mathbf{k})$
$= -2 + 6*\mathbf{i} + 7+\mathbf{j} + 3*\mathbf{k}$

As you can see, the writing of the imaginary coefficients is getting rather tedious, and we could have just written

<3,4,5,6> + <-5,2,2,-3> = <-2,6,7, 3>.

However, we have to be careful, because although it works for addition and subtraction, for multiplication we need to recall that the last three components are complex. To avoid oversimplification, we need to keep the quaternions in a form that at least keeps the real and imaginary parts separate.

Additive Inverse and Identity

The additive inverse of any quaternion **q** is the number that, when added to **q**, results in zero. This is surprisingly just –q:

Given:

$\mathbf{q} = q_0 + \mathbf{q}_v$

The additive inverse is just

$-\mathbf{q} = -q_0 - \mathbf{q}_v$

because

$\mathbf{q} - \mathbf{q} = (q_0-q_0)+ (\mathbf{q}_v- \mathbf{q}_v) = 0 + 0*i + 0*j + 0*j$

The additive identity, or "0" in quaternion math, must be

$\mathbf{q} = 0 + 0*\mathbf{i} + 0*\mathbf{j} + 0*\mathbf{k}$ = <0,0,0,0>.

Playrix Ex. 1023, page 353

07 0672318350 CH04 05•02•2003 2:36 PM Page 329

Quaternion Multiplication

Addition and subtraction is always easy, huh? It's multiplication that ruins the party every time! And this is no exception. Because quaternions are nothing more than hypercomplex numbers based on real numbers with imaginary coefficients, we should be able to just multiply them out, taking into consideration the imaginary coefficients in the products and keeping track of them. Let's just try it and see what happens.

Given

$$\mathbf{p} = p_0 + p_1\mathbf{i} + p_2\mathbf{j} + p_3\mathbf{k} = p_0 + \mathbf{p}_v$$
$$\mathbf{q} = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} = q_0 + \mathbf{q}_v$$

then

$$\mathbf{p}*\mathbf{q} = (p_0 + p_1\mathbf{i} + p_2\mathbf{j} + p_3\mathbf{k}) * (q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k})$$
$$= p_0*q_0 +$$
$$p_0*q_1\mathbf{i} + p_1*q_2\mathbf{j} + p_2*q_3\mathbf{k} +$$
$$p_1\mathbf{i}*q_0 + p_1\mathbf{i}*q_1\mathbf{i} + p_1\mathbf{i}*q_2\mathbf{j} + p_1\mathbf{i}*q_3\mathbf{k} +$$
$$p_2\mathbf{j}*q_0 + p_2\mathbf{j}*q_1\mathbf{i} + p_2\mathbf{j}*q_2\mathbf{j} + p_2\mathbf{j}*q_3\mathbf{k} +$$
$$p_3\mathbf{k}*q_0 + p_3\mathbf{k}*q_1\mathbf{i} + p_3\mathbf{k}*q_2\mathbf{j} + p_3\mathbf{k}*q_3\mathbf{k}$$

If you have a good eye, you should see some structure to this product—maybe a cross-product here and there, and a dot product? Keep that in mind while we collect terms and use Equation 4.30 to simplify the imaginary product terms:

$$= p_0*q_0 +$$
$$p_0*q_1\mathbf{i} + p_1*q_2\mathbf{j} + p_2*q_3\mathbf{k} +$$
$$p_1*q_0\mathbf{i} + p_1*q_1\mathbf{i}^2 + p_1\mathbf{i}*q_2\mathbf{j} + p_1\mathbf{i}*q_3\mathbf{k} +$$
$$p_2\mathbf{j}*q_0 + p_2*q_1\mathbf{j}\mathbf{i} + p_2*q_2\mathbf{j}^2 + p_2*q_3\mathbf{j}\mathbf{k} +$$
$$p_3*q_0\mathbf{k} + p_3*q_1\mathbf{k}\mathbf{i} + p_3*q_2\mathbf{k}\mathbf{j} + p_3*q_3\mathbf{k}^2$$

At this point, there are a lot of ways to format these products, but I will stop here and show you this equation.

EQUATION 4.31 Formula for Quaternion Products

Given

$$\mathbf{p} = p_0 + p_1\mathbf{i} + p_2\mathbf{j} + p_3\mathbf{k} = p_0 + \mathbf{p}_v$$
$$\mathbf{q} = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} = q_0 + \mathbf{q}_v$$

then

$$\mathbf{r} = \mathbf{p}*\mathbf{q} = (p_0*q_0 - (\mathbf{p}_v \cdot \mathbf{q}_v)) + (p_0*\mathbf{q}_v + q_0*\mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v)$$
$$= r_0 + \mathbf{r}_v$$

Playrix Ex. 1023, page 354

07 0672318350 CH04 05•02•2003 2:36 PM Page 330

NOTE

The *x* operator is the standard vector cross-product, and is calculated on the vector imaginary part of the quaternion as if the imaginary part was a standard 3-tuple vector.

And because dot products always result in scalars and cross products result in vectors, the first term (p_0*q_0 – ($\mathbf{p}_v$. $\mathbf{q}_v$)) is the real part r_0, and the term (p_0*$\mathbf{q}_v$ + q_0*$\mathbf{p}_v$ + $\mathbf{p}_v$ x $\mathbf{q}_v$) is the vector or imaginary part $\mathbf{r}_v$. I have to admit that quaternion multiplication is a very ugly thing in mathematics!

Also, note that the *multiplicative identity* (the analog of "1" in quaternion math) is

$\mathbf{q}_1 = 1 + 0*\mathbf{i} + 0*\mathbf{j} + 0*\mathbf{k}$

This is true because any $\mathbf{q} * \mathbf{q}_1 = \mathbf{q} = \mathbf{q}_1 * \mathbf{q}$.

Quaternion Conjugate

Computing the conjugate of a quaternion $\mathbf{q}$, $\mathbf{q}^*$ is accomplished in the same manner as with complex numbers: You simply invert the sign of the imaginary component $\mathbf{q}_v$.

EQUATION 4.32 Computation of Complex Conjugate

Given

$\mathbf{q} = q_0 + q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k} = q_0 + \mathbf{q}_v$

The complex conjugate is computed by inverting the sign of the imaginary part:

$\mathbf{q}^* = q_0 - q_1*\mathbf{i} - q_2*\mathbf{j} - q_3*\mathbf{k} = q_0 - \mathbf{q}_v$

And take a look at the product of $\mathbf{q}$ and $\mathbf{q}^*$—it's rather interesting (using Equation 4.31 to perform the multiplication):

EQUATION 4.33 The Quaternion-Conjugate Product

$$\begin{aligned}\mathbf{q} * \mathbf{q}^* &= (q_0 + \mathbf{q}_v) * (q_0 + (-\mathbf{q}_v)) \\ &= q_0*q_0 - (\mathbf{q}_v \cdot (-\mathbf{q}_v)) + q_0*\mathbf{q}_v + q_0*(-\mathbf{q}_v) + (\mathbf{q}_v \times (-\mathbf{q}_v)) \\ &= q_0^2 + q_1^2 + q_2^2 + q_3^2 + (q_0*\mathbf{q}_v - q_0*\mathbf{q}_v) + \mathbf{(0)} \\ &= q_0^2 + q_1^2 + q_2^2 + q_3^2\end{aligned}$$

Interestingly, the product is simply the square of each term. This property will come in handy when we deal with computing the norm and inverse of a quaternion.

Norm of a Quaternion

The norm of a quaternion is computed in the same manner as a complex number:

Playrix Ex. 1023, page 355

07 0672318350 CH04 05•02•2003 2:36 PM Page 331

EQUATION 4.33 Norm of a Quaternion

Given

$$\mathbf{q} = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} = q_0 + \mathbf{q}_v$$

then

$$|\mathbf{q}| = \text{sqrt}(q_0^2+q_1^2+q_2^2+q_3^2) = \text{sqrt}(\mathbf{q} * \mathbf{q}^*)$$

And of course,

$$|\mathbf{q}|^2 = (q_0^2 + q_1^2 + q_2^2 + q_3^2) = (\mathbf{q} * \mathbf{q}^*)$$

MATH

Note that $(\mathbf{q} * \mathbf{q}^*) = (\mathbf{q}^* * \mathbf{q})$. The product of a quaternion and its conjugate is order-independent, but in general is $\mathbf{q}*\mathbf{p} \neq \mathbf{p}*\mathbf{q}$. Also, note that $\mathbf{q} + \mathbf{q}^* = 2*q0$.

4

Multiplicative Inverse

The multiplicative inverse of a quaternion is of special importance to us, because it can be used to simplify quaternion rotation. In fact, everything we have learned up until now is going to come to the single focus of rotating a vector using quaternions, and the inverse is needed for this operation. So put on your thinking cap and check it out…

Given a quaternion $\mathbf{q}$, we want to find another quaternion $\mathbf{q}^{-1}$ such that the following statement is true:

$$\mathbf{q}*\mathbf{q}^{-1} = 1 = \mathbf{q}^{-1}*\mathbf{q}$$

Don't blink and watch this…

Let's multiply each side by the complex conjugate $\mathbf{q}^*$:

$$(\mathbf{q} * \mathbf{q}^{-1}) * \mathbf{q}^* = 1 = (\mathbf{q}^{-1} * \mathbf{q}) * \mathbf{q}^* = \mathbf{q}^*$$

Because the products in parentheses are 1, we know that 1 multiplied by the conjugate and the conjugate multiplied by 1 are both the conjugate, so we haven't changed anything or created a false statement. Alright, now's the cool part—let's stare at this a minute and see whether we can figure out what the inverse is. Do you see it? Let's look at the right side alone:

$$(\mathbf{q}^{-1} * \mathbf{q}) * \mathbf{q}^* = \mathbf{q}^*$$

or

$$\mathbf{q}^{-1} * (\mathbf{q} * \mathbf{q}^*) = \mathbf{q}^*$$

Playrix Ex. 1023, page 356

07 0672318350 CH04 05•02•2003 2:36 PM Page 332

But

$$(\mathbf{q} * \mathbf{q}^*) = |\mathbf{q}|^2$$

The final result is as seen in Equation 4.34a.

Equation 4.34a Inverse of a Quaternion

$$\mathbf{q}^{-1} = \mathbf{q}^* / |\mathbf{q}|^2$$

Furthermore, if $\mathbf{q}$ is a unit quaternion, then $|\mathbf{q}|^2 = 1$ and we can further simplify the inverse to Equation 4.34b.

Equation 4.34b Inverse of a Unit Quaternion

$$\mathbf{q}^{-1} = \mathbf{q}^*$$

Pretty cool, huh? The preceding equation is the whole reason that makes using quaternions to perform rotations even possible. Thus, most of the time we will assume that all quaternions are unit quaternions, so we can use the preceding equation without a problem.

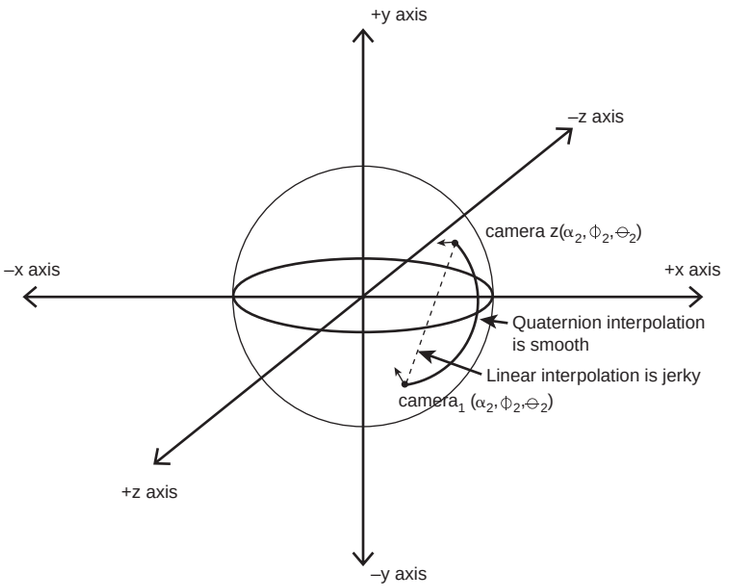


FIGURE 4.44 Interpolation from one orientation to another.

Playrix Ex. 1023, page 357

07 0672318350 CH04 05•02•2003 2:36 PM Page 333

Applications of Quaternions

After all that work learning about quaternions, they probably seem just like interesting mathematical objects with little or no real use to us. However, the fact is that quaternions are very useful for a couple of functions in 3D graphics—rotation and the interpolation of rotations from one to another, as shown in Figure 4.44. In the figure you see two camera directions defined by two sets of angles relative to the x-y-z axis $camera_1 = (\alpha_1, \phi_1, \theta_1)$ and $camera_2 = (\alpha_2, \phi_2, \theta_2)$. How we point the camera in those directions is unimportant, but the point is, what if we want to smoothly interpolate from the orientation of $camera_1$ to $camera_2$? We could linearly interpolate based on the angles, but a number of problems can occur when you do this, such as jerky motion and the loss of a degree of freedom when the camera happens to align on an axis.

The point is that quaternions, by their 4D nature, can handle this problem much more elegantly than standard angles and rotation matrices; they are a bit slower, but worth it. We aren't going to use quaternions to perform standard 3D rotations and manipulations, but we will use them for advanced camera work, as manipulating a 3D camera with quaternions turns out to be more intuitive than using angles. I'm getting way ahead of myself here, so let's just learn how to perform a rotation of a vector with quaternions, and then when we get to camera work, we will revisit them and learn more about using quaternions to interpolate camera orientations and so forth. Right now, I just want you to see the math.

Quaternion Rotation

At some point you (as a mathematician) might wonder what operators can be represented by quaternion transformations—in other words, can they do anything interesting? The answer is yes. One of those interesting and useful operations is the capability to rotate a vector **v**. I would love to show you all the math to arrive at that result, but as usual, I am under both time and page constraints, and it's only interesting to a math maniac, so you're not missing anything. The end result is that given a vector $\mathbf{v} = \langle x,y,z \rangle$, which in quaternion form we will call $\mathbf{v}_q = \langle 0,x,y,z \rangle$ in 3D (we use a dummy $0 = q_0$, so that we can use **v** as a quaternion), and a unit quaternion **q**, the following operations will rotate $\mathbf{v}_q$:

EQUATIONS 4.35 Quaternion Rotation

Right-handed system:

a. $\mathbf{v}_q' = \mathbf{q}^* * \mathbf{v}_q * \mathbf{q}$, clockwise rotation

b. $\mathbf{v}_q' = \mathbf{q} * \mathbf{v}_q * \mathbf{q}^*$, counter-clockwise rotation

Left-handed system:

c. $\mathbf{v}_q' = \mathbf{q} * \mathbf{v}_q * \mathbf{q}^*$, clockwise rotation

d. $\mathbf{v}_q' = \mathbf{q}^* * \mathbf{v}_q * \mathbf{q}$, counter-clockwise rotation

Playrix Ex. 1023, page 358

07 0672318350 CH04 05•02•2003 2:36 PM Page 334

MATH

We can use $\mathbf{q}^*$ in this operator only for unit quaternions; otherwise, you must use the full inverse of $\mathbf{q}$.

So $\mathbf{v}_q$ is a vector encoded in a quaternion with $q_0 = 0$, and $\mathbf{q}$ is a quaternion, but what exactly does $\mathbf{q}$ represent? What's its relationship to the x-y-z axis and $\mathbf{v}_q$? You're going to love this—$\mathbf{q}$ defines both the axis of rotation and the angle θ to rotate about the axis! Of course, the result $\mathbf{v}_q'$ is technically a 4D vector or a quaternion. However, the first component, q_0, will always be zero, so we just throw it away and think of the last three elements as just a vector in 3D representing the original vector $\mathbf{v}$ after rotation.

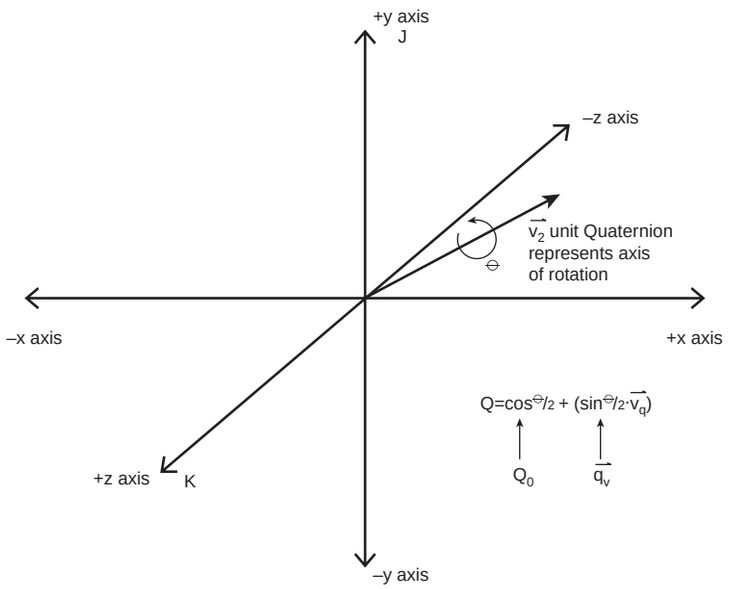


FIGURE 4.45 A geometrical interpretation of a rotation quaternion.

If you have ever tried to rotate a 3D object around an arbitrary axis, you know it's not that easy—but now it is. The axis that the quaternion $\mathbf{q}$ defines along with the angle isn't that obvious, but it's not that bad. Referring to Figure 4.45, we see that for a given unit quaternion $\mathbf{q} = q_0 + \mathbf{q}_v$, the axis of rotation is just the line defined by the vector part $\mathbf{q}_v$, and the angle of rotation θ is encoded in q_0 using the following transforms:

Playrix Ex. 1023, page 359

07 0672318350 CH04 05•02•2003 2:36 PM Page 335

EQUATION 4.36 Conversion of Axis and Angle to Quaternion

$\mathbf{q} = \cos(\theta/2) + \sin(\theta/2)*\mathbf{v}_q$

Thus,

$q_\theta = \cos(\theta/2)$ and $\mathbf{q}_v = \sin(\theta/2)* \mathbf{v}_q$

NOTE

Of course, $\mathbf{q}_v$ must be a unit vector itself, so that $\mathbf{q}$ remains a unit quaternion.

What if you don't have a vector that you want to rotate around, but have the standard Euler rotation angles? You can then create a quaternion from the angles by generating one of the products:

$$\begin{aligned}\mathbf{q}_{final} &= \mathbf{q}_{x\theta}*\mathbf{q}_{y\theta}*\mathbf{q}_{z\theta}\\ &= \mathbf{q}_{x\theta}*\mathbf{q}_{z\theta}*\mathbf{q}_{y\theta}\\ &= \mathbf{q}_{y\theta}*\mathbf{q}_{x\theta}*\mathbf{q}_{z\theta}\\ &= \mathbf{q}_{y\theta}*\mathbf{q}_{z\theta}*\mathbf{q}_{x\theta}\\ &= \mathbf{q}_{z\theta}*\mathbf{q}_{x\theta}*\mathbf{q}_{y\theta}\\ &= \mathbf{q}_{z\theta}*\mathbf{q}_{y\theta}*\mathbf{q}_{x\theta} \text{ <-most common transform}\end{aligned}$$

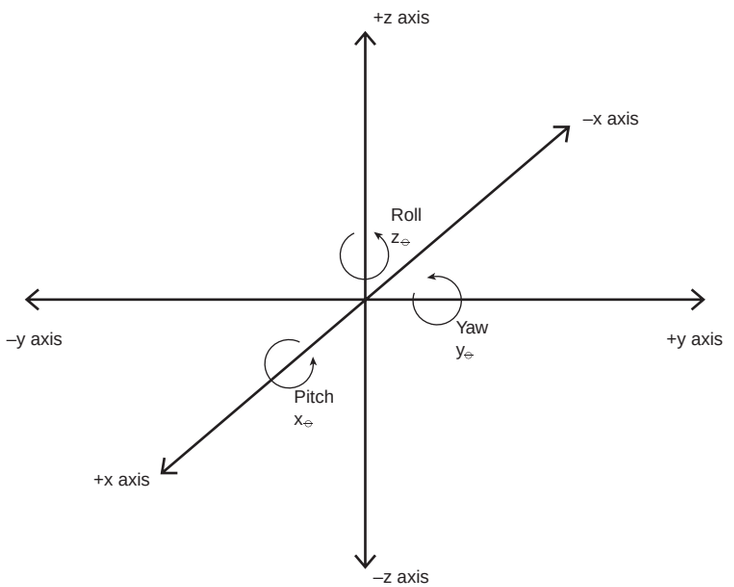


FIGURE 4.46 Yaw, pitch, and roll.

Playrix Ex. 1023, page 360

07 0672318350 CH04 05•02•2003 2:36 PM Page 336

where x_θ refers to the *pitch* (angle parallel to the x-axis), y_θ refers to the *yaw* (angle parallel to the y-axis), and z_θ refers to the *roll* (angle parallel to the z-axis), as shown in Figure 4.46. Most people use the last transform most commonly in 3D engines, but all are just as valid. In any case, after you have the order of transforms, you can plug in the following formulas for the quaternions $\mathbf{q}_{x\theta}$, $\mathbf{q}_{y\theta}$, $\mathbf{q}_{z\theta}$ and perform the multiplication to arrive at $\mathbf{q}_{final}$.

EQUATION 4.37 Euler Angles to Quaternion

$$\mathbf{q}_{x\theta} = \cos(x_\theta/2) + \sin(x_\theta/2)*\mathbf{i} + 0*\mathbf{j} + 0*\mathbf{k}$$
$$q_\theta = \cos(x_\theta/2),\ \mathbf{q}_v = \langle \sin(x_\theta/2), 0, 0 \rangle$$

$$\mathbf{q}_{y\theta} = \cos(y_\theta/2) + 0*\mathbf{i} + \sin(y_\theta/2)*\mathbf{j} + 0*\mathbf{k}$$
$$q_\theta = \cos(y_\theta/2),\ \mathbf{q}_v = \langle 0, \sin(y_\theta/2), 0 \rangle$$

$$\mathbf{q}_{z\theta} = \cos(z_\theta/2) + 0*\mathbf{i} + 0*\mathbf{j} + \sin(z_\theta/2)*\mathbf{k}$$
$$q_\theta = \cos(z_\theta/2),\ \mathbf{q}_v = \langle 0, 0, \sin(z_\theta/2) \rangle$$

So to rotate a vector **v** using the Euler form, you are really doing the following for the example of roll, yaw, and pitch:

$$\mathbf{v}_q'(0,x',y',z') = (\mathbf{q}_{z\theta}*\mathbf{q}_{y\theta}*\mathbf{q}_{x\theta})*\mathbf{v}_q*(\mathbf{q}_{z\theta}*\mathbf{q}_{y\theta}*\mathbf{q}_{x\theta})^*$$

where $\mathbf{v}_q = \langle 0,x,y,z \rangle$, the initial vector or point we want to rotate cast into quaternion form.

Of course, you only need to compute the product in parentheses once. And because each of the factors $\mathbf{q}_{x\theta}$, $\mathbf{q}_{y\theta}$, $\mathbf{q}_{z\theta}$ is of the form below:

$$\mathbf{q}_{(i,j,k)} = \cos(\theta/2) + \sin(\theta/2)*(\mathbf{i},\mathbf{j},\mathbf{k})$$

The product of terms of this form have a lot of cancellations—thus the product $\mathbf{q}_{z\theta}*\mathbf{q}_{y\theta}*\mathbf{q}_{x\theta}$,for example, ends up looking like this:

$$q_0 = \cos(z_{\theta/2})*\cos(y_{\theta/2})*\cos(x_{\theta/2}) + \sin(z_{\theta/2})*\sin(y_{\theta/2})*\sin(x_{\theta/2})$$
$$q_1 = \cos(z_{\theta/2})*\cos(y_{\theta/2})*\sin(x_{\theta/2}) - \sin(z_{\theta/2})*\sin(y_{\theta/2})*\cos(x_{\theta/2})$$
$$q_2 = \cos(z_{\theta/2})*\sin(y_{\theta/2})*\cos(x_{\theta/2}) + \sin(z_{\theta/2})*\cos(y_{\theta/2})*\sin(x_{\theta/2})$$
$$q_3 = \sin(z_{\theta/2})*\cos(y_{\theta/2})*\cos(x_{\theta/2}) - \cos(z_{\theta/2})*\sin(y_{\theta/2})*\sin(x_{\theta/2})$$

NOTE

Note that each quaternion in the products can be thought of as an individual rotation operation, but it takes much less work to multiply quaternions than matrices. Therefore, if you find yourself doing a lot of matrix multiplications in a loop, you might be able to optimize it by converting to quaternions—doing all the matrix operations in quaternion form and then converting back.

Playrix Ex. 1023, page 361

07 0672318350 CH04 05•02•2003 2:36 PM Page 337

The conversions from matrix to quaternion and quaternion to matrix are much more complex, and I don't have time to derive them here. However, I will hold off on them until we once again get to the camera stuff, so that it makes more sense to you within the context of something tangible. Also, we will cover interpolation of quaternions later in the camera animation coverage.

Basic Calculus

I never thought I would have to cover basic calculus in a game book, but these days you need to be a damn mathematician to write games! Calculus is nothing more than a form of mathematics invented independently by both Sir Isaac Newton and Baron Gottfried Wilhelm Leibniz in the 17th century—brainiacs deluxe! "The Calculus" (as it's referred to) was invented as a tool to help solve problems in motion, change, and dynamic systems that standard static algebra of the time couldn't handle. Thus, calculus is the mathematics of change, or systems in flux.

As far as we are concerned as game programmers, there is no way you can even think of creating a state-of-the-art 3D game these days without avoiding calculus and differential equations. 3D graphics used to be manageable with linear algebra and a good grasp of matrix mathematics and trig, but now the engines are getting more and more complex, and people want higher order surfaces, like spline and Bezier patches, and so forth. However, that's not the problem—we could all live without rendered curves in games, but lots of camera problems need splines and curves as well, and the mathematics of these objects sometimes take a little calculus to manipulate. However, the *real* problem that makes calculus an absolute necessity is *physics*.

Physics is why calculus was invented! These days friction and a little collision detection isn't going to cut it—people want to see real physics, and that means real math and real models. There is no way to avoid using calculus, so you might as well start learning it right now.

Of course, I can't hope to teach you four years of university math in 20 pages, but I am going to at least show you a couple of concepts. When I use them, you might not understand exactly what I am doing, but you won't be completely lost. I realize that many people reading this book will be in high school and many will be graduate students, but I don't want to leave anyone out in the cold (the biggest complaint I get about game programming books is that people want more math). I'm going to give you a very brief, very loose, and very abridged treatise on the most important things in calculus, and you can pick up a calculus book on your own if you are interested in further reading. I suggest *Calculus* by Howard Anton, or *Calculus* by Thomas & Finney—they rock.

The Concept of Infinity

The first topic I want to discuss might not change your whole life, but it's real, it works, and it's a mathematical reality—*infinity*, represented by the symbol ∞. Infinity is one of

Playrix Ex. 1023, page 362

07 0672318350 CH04 05•02•2003 2:36 PM Page 338

those things that we've all heard about, but can't really put our finger on. In calculus however, we love infinity and have no problem with it. In fact, there are all kinds of infinities; small, large, negative, and positive. These are all mathematical tools, and don't need to represent things that are physical, but many times they do.

In any case, I want you to be able to work with infinity because we might need it, and later we might get results that are infinite, which is perfectly fine in higher mathematics. We can start with infinity, or perform a mathematical operation that results in infinity. For example, take a look at the following quotient:

```
      1
q = ----
      y
```

That's a perfectly reasonable thing to do in a computer program, but what happens if y becomes very small? Let's see:

```
1/1 = 1, 1/.1 = 10, 1/.01 = 100, 1/.001 = 1000,…
```

As you can see, as we let y get smaller and smaller, the quotient of 1/y gets larger and larger. You might say, I can make the numerator 100,000,000,000. And I say, so what? I can make the denominator 10 times that and the result will be a small number. The point is that as the denominator gets very small, the quotient gets larger and larger without boundary. At some point, the denominator will become zero, but what will the quotient become? The answer is plus infinity.

```
x/0 = +∞, +infinity
```

It's irrelevant what x is. Also, interestingly enough, the following also holds true (sorta):

```
 x
--- = 0
 ∞
```

Which makes sense, because if you look at the following sequence, you can see it:

```
1/1 = 1, 1/10 = .1, 1/100 = .01, 1/1000 = .001, …
```

As the denominator gets larger and larger, the quotient gets smaller and smaller, and in the end the result will be zero.

The last question you might ask is what happens if both the numerator and the denominator go to infinity? Well, it depends on which one gets there faster. Let's think about it with common sense. Take a look at this:

```
      500*x
y = ----------
       x*x
```

Playrix Ex. 1023, page 363

07 0672318350 CH04 05•02•2003 2:36 PM Page 339

The question is, what happens when x approaches infinity? Well, in the numerator we have 500*x, and in the denominator we have x*x. Here's a short run of the numerator and denominator for $x \in (0,1,2,3,4,5,6,7,8,9,10)$:

TABLE 4.6 A Run of Numbers in the Quotient $500*x/x^2$

x	Numerator (500*x)	Denominator(x*x)	Quotient
0	0	0	Indeterminate
1	500	1	500
2	1000	4	250
3	1500	9	166.66
4	2000	16	125
5	2500	25	100
6	3000	36	83.33
7	3500	49	71.42
8	4000	64	62.5
9	4500	81	55.55
10	5000	100	50

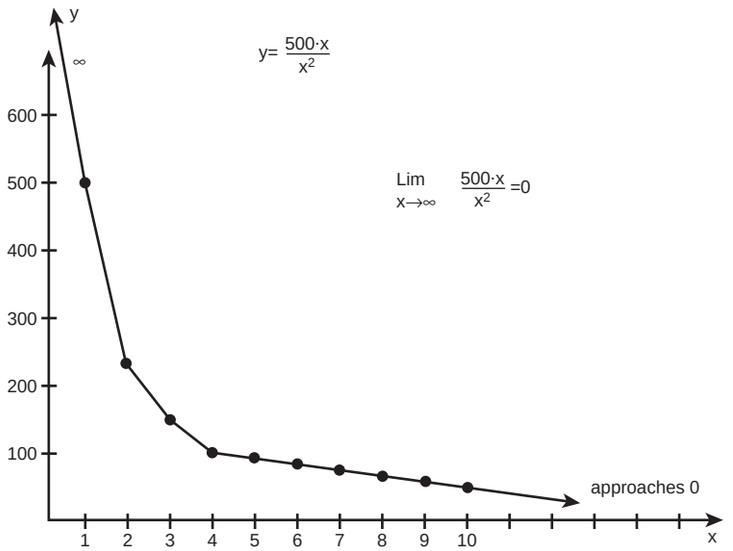


FIGURE 4.47 A graph of the quotient $500*x/x^2$.

Also, take a look at the graph in Figure 4.47. Although it seems that the numerator is moving very quickly, the denominator is quickly catching up, and thus the quotient is getting smaller and smaller. In fact, the end result will be zero. So does common sense seem right? Well, at first glance, if you didn't run the sequence, you might say that the

Playrix Ex. 1023, page 364

07 0672318350 CH04 05•02•2003 2:36 PM Page 340

numerator is much larger, and it is for small x. But as x gets large, the x*x is much larger than 500*x, so the end result is that the rate of increase of the denominator is greater than the numerator. Hence, the result will be zero in this case. We could have figured this out using a little algebra, actually.

If we look at the original quotient and simplify it, we can see the result of zero much easier.

```
     500*x
y = ------- = 500/x
      x*x
```

According to our first rule, we know that as x approaches infinity, anything divided by x approaches zero. This whole discussion leads into my next topic—limits.

Limits

The subject of limits is very large, but basically they are a formalization of what we have been doing in the previous section. In general, a *limit* is a mathematical statement that helps us write down a concept that has a changing variable that approaches a fixed constant. This is the very point of calculus—being able to write down in a consistent manner changing mathematical concepts. In any case, here's the general form of a limit statement:

```
Lim  f(x) = L
x->a
```

This is read "The limit of f(x) as x approaches a is equal to L." Computing limits are fairly simple for the more straightforward ones, but of course, they can get very complex. However, in our case, we will just stick to very simple ones. In essence, to compute a limit, you simply let x (or whatever the variable is) approach the limiting value (a in this case) and read off the result. This is best shown with examples:

Example 1: Simple limit

```
Lim 2*x = ?
x->3
```

As x approaches 3, nothing weird happens with f(x). Thus we can simply replace x with 3, and the limit is equal to

```
Lim 2*x = 2*(3) = 6
x->3
```

Example 2: Indeterminate limit

```
Lim x*x/3*x
x->0
```

Playrix Ex. 1023, page 365

07 0672318350 CH04 05•02•2003 2:36 PM Page 341

Now this one is a bit tricky. If we let x approach 0, then we get

```
Lim 0*0/3*0 = ?
x->0
```

which is indeterminate (0/0 is not computable). We need to see whether we can simplify the limit first, so we can get a better grasp on what's happening:

```
Lim x*x/3*x
x->0
```

Canceling out the x terms in numerator and denominator gives us

```
Lim x/3 = 0
x->0
```

which clearly equals zero, because 0/3 = 0.

Example 3: Infinite limit

```
Lim 1/(5*x*x + 2*x) = ?
x->∞
```

We have seen this before, when we were talking about infinity. In this case, the bottom line is that as x gets larger and approaches infinity, (5*x*x + 2*x) gets larger and approaches infinity even faster. Thus the quotient goes to 1/∞, which is equal to zero.

NOTE

Limits can be useful, especially when you are trying to figure out what some expression will end up being as some variable approaches another.

Summation and Finite Series

The next topic I want to discuss isn't really hard-core calculus, but it's a nice way to ease into the subject of integrals, and has a real calculus feel to it. Many times when solving problems that are iterative or recursive in nature, we see sequences of numbers or terms that have patterns. For example, take a look at the following mathematical sequence or summation:

```
1+2+3+4+..n
```

Before we try to compute this, let's try and find a more compact way to write it using *sigma notation* (using the ∑ symbol). The preceding sequence has a lower limit (1), an upper limit (n), and a rule. In this case, the rule is at each step simply add the next number in the sequence. It's really just a for loop:

Playrix Ex. 1023, page 366

07 0672318350 CH04 05•02•2003 2:36 PM Page 342


```
for (int count = 1; sum =0; count <=100; count++)
  sum+=count; // <-the rule
```

This C/C++ code is written mathematically like this:

$$\text{sum} = \sum_{n=1}^{1000} n$$

which is read something like "sum equals the rule n as n varies from 1 to 1000." In general, the sigma or summation notation looks like this:

$$\sum_{n=a}^{b} f(n)$$

where a <= b (but not necessarily), a is called the "lower limit," and b is called the "upper limit."

Now, back to computing the answer. If n was equal to 1000, how long do you think it would take to add the numbers 1 to 1,000 up? Well, there's a formula for this (and many other sums) discovered by the great mathematician Karl Gauss:

EQUATION 4.38a The Sum of the First n Integers

$$\sum_{k=1}^{n} k = 1+2+3+\ldots+n = n*(n+1)/2$$

Plugging in `1000`, we get

```
1000*(1000+1)/2 = 500500
```

If you think that's impressive, Gauss was five years old when he figured that one out! Of course, throughout the years mathematicians have figured out many *closed forms* for various sequences. Here are two more that come in handy:

EQUATION 4.38b The Sum of the First n Squares

$$\sum_{k=1}^{n} k^2 = 1^2+2^2+3^2+\ldots+n^2 = n*(n+1)*(2*n+1)/6$$

EQUATION 4.38c The Sum of the First n Cubes

$$\sum_{k=1}^{n} k^3 = 1^3+2^3+3^3+\ldots+n^3 = (n*(n+1)/2)^2$$

Playrix Ex. 1023, page 367

07 0672318350 CH04 05•02•2003 2:36 PM Page 343

MATH

Being able to work with series and sums helps with *asymptotic analysis* (see *Tricks of the Windows Game Programming Gurus*), which is the analysis of algorithms and deals with computing the runtime of inner loops and so on. So this is good stuff for 3D graphics.

Now that you have a couple of formulas under your belt, let's see whether we can figure out some sums that are a little bit different. For example:

$$\sum_{k=1}^{n} 3*k = ?$$

The first thing we do is write out a few terms to see the pattern, and find out whether we can manipulate it into one of our formulas and solve it.

$$\sum_{k=1}^{n} 3*k = 3*1 + 3*2 + 3*3 + \ldots + 3*n$$

Looks like 3 is always there, so let's factor it out of the sigma sign like this:

$$\sum_{k=1}^{n} 3*k = 3*1 + 3*2 + 3*3 + \ldots + 3*n$$

$$= 3*\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$$

$$= 3*n*(n+1)/2$$

Or in general, for any scalar *a*:

$$\sum_{k=1}^{n} a*k = a*n*(n+1)/2$$

We can do this stuff all day long and come up with all kinds of cool summation formulas—it's good exercise for both your brain and your hands to tell you the truth! Now let's move on to infinite series that are more appropriate for some of the physics calculations you might make in your 3D programming career.

Playrix Ex. 1023, page 368

07 0672318350 CH04 05•02•2003 2:36 PM Page 344

Infinite Series

Finite sums are things that are very tangible—they end somewhere. But there are mathematical sequences that go on forever, but aren't infinite! How can something go on forever, but not be infinite? Well, look at this, for example:

```
0 + 0 + 0 + …
```

An infinite sum of zeros is definitely still zero. This is the theme of the next section.

There are many times when a complex system can run forever, but its height, energy, level, and so on are changing each time. These systems are real, and arise in nature and mathematics, so being able to come up with closed sums is very useful to us. Let's begin with some notation. A given series might look like this:

Domain:	0	1	2	3 …	n
	↓	↓	↓	↓	↓
Range:	0	5	10	15	5*n

In this case, the domain is the positive integers 0, 1, 2 … and the range is also the positive integers. The rule is multiply the integer by 5.

The question is, what is the sum of this infinite series? Common sense tells us that the series keeps getting larger and larger, and thus must grow unbounded, and hence the sum is infinity. However, there are other series that do converge to a finite number. For example, series of the following form *can* converge to a finite value, if the values of *a* and *r* satisfy certain constraints:

$$\sum_{n=1}^{\infty} a*r^{(n-1)} = a + a*r + a*r^2 + a*r^3 + \ldots + a*r^n + \ldots$$

where *a* and *r* are real numbers, and $a \neq 0$. Infinite series of this kind are called *geometric series*. Here's an example:

Example:

```
a = 5, r = 1/2
```

$$\sum_{n=1}^{\infty} a*r^{(n-1)} = a + a*r + a*r^2 + a*r^3 + \ldots + a*r^n + \ldots$$

$$= 5 + 5*1/2 + 5*(1/2)^2 + 5*(1/2)^3 + \ldots$$

$$= 5 + 5/2 + 5/4 + 5/8 + \ldots$$

Playrix Ex. 1023, page 369

07 0672318350 CH04 05•02•2003 2:36 PM Page 345

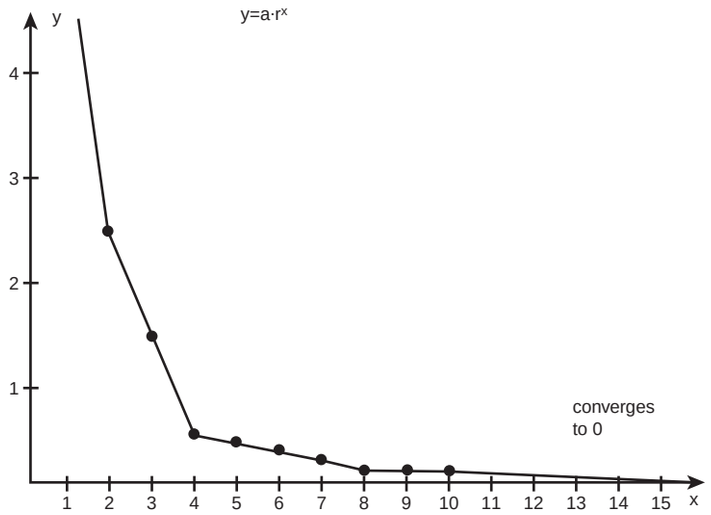


FIGURE 4.48 A plot of the geometric series a = 5, r = 1/2.

We can definitely see that the series is getting smaller and smaller, and this is confirmed by the plot of points in Figure 4.48. In fact, when n reaches infinity, the terms will approach zero. So, there does seem to be good reason to believe there is a finite sum—and of course there is!

EQUATION 4.39 Sum of a General Geometric Series

$$\sum_{n=1}^{\infty} a*r^{(n-1)} = a + a*r + a*r^2 + a*r^3 + \ldots + a*r^n + \ldots$$

$$\text{the sum} = a/(1-r), \text{ for } |r| < 1$$

That's got to be one of the coolest formulas in the world. It tells us how to compute the sum of an infinite collection of expressions in a closed form—math rocks! Using the formula on our series from the previous example with a = 5, and r = 1/2, |r| is less than 1, so it's cool to use, and the sum is

$$a/(1-r) = 5/(1/2) = 10.$$

Of course, there are many kinds of infinite series, but I leave it to you to explore more …

Derivatives

Now we are finally ready to enter the twilight zone—real calculus. However, calculus is really baby math in the scheme of things, as it's the first course in college mathematics. On the other hand, I have not known many engineers that need to use it on a daily basis,

Playrix Ex. 1023, page 370

07 0672318350 CH04 05•02•2003 2:36 PM Page 346

but game programming is turning out to be a field where I see the need for it in the near future. This section and the next are going to be the briefest explanation of derivatives and integrals in the world, but you will at least have an idea of what's going on when I use them later in the book. Furthermore, I am going to be very loose with definitions and rigor.

For example, in high school they teach you that atoms are made of electrons, protons, and neutrons, and that's it. Of course, that's not really true. Furthermore, electrons do NOT orbit the nuclei of the atom. In general, they are a probability cloud. In the same vein, my math here is going to oversimplify reality, but be correct enough for our usage, or rather, detailed enough.

If you recall, the idea of calculus is to be able to discuss and manipulate mathematical objects that change as a function of time (or another variable); that is, functions that are not static. At the most basic level of this notion is the derivative function. The *derivative* of a function is the instantaneous rate of change of that function. For example, suppose you have an object in a game, and its position is described by the following formula, as shown in Figure 4.49:

```
x = x0 + vel*t
```

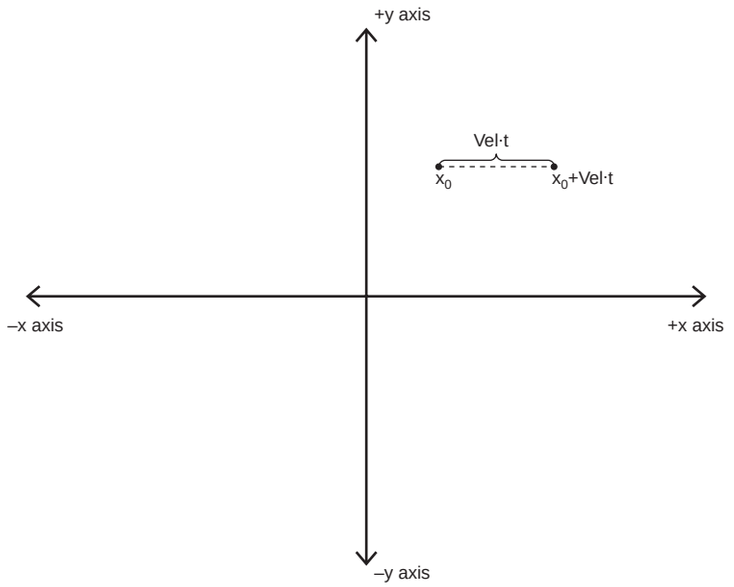


FIGURE 4.49 An object moving based on the motion model (x = x0 + vel*t).

This obviously means that the position x at any time is x0 plus the factor vel*t, where t could be time, or some artificial variable, like the current frame. In either case, we might

Playrix Ex. 1023, page 371

07 0672318350 CH04 05•02•2003 2:36 PM Page 347

ask the question, given the formula for *x*, what is the instantaneous rate of change of x? The answer to this is in the definition of x, because we know that with each increment of t, x will change by vel*dt, or velocity*1. Thus, the instantaneous rate of change is the rate factor vel itself. But what if we weren't so darn smart? How could we mathematically determine this without having seen this so many times that we just know? The answer is that the derivative tells us.

If we differentiate x with respect to t, the result is the rate of change of x with respect to t. Sounds pretty circular, huh? In other words, the derivative of the position *is* the velocity. Here's what taking the derivative looks like:

```
d/dt * x(t)
```

which means take the derivative of x with respect to t.

```
d/dt * x(t) = vel
```


Convenient, huh? Almost magical! At this point, if you are thinking "What the heck?" that's okay. Let's see how all this works.

Definition of Derivative

Later I will show you a couple of different notations for derivatives, but for now let's just forget about that and focus on what it is. If you take look at Figure 4.50, you see a curve that's a function of x, f(x). Also, you see two points on the curve: one labeled **a**, which has the value x0, and **b**, which has the value x0+h (h is just a number), and between the two points you see a tangent line with a slope. The slope of this tangent line is shown in Equation 4.40.

EQUATION 4.40 Definition of the Tangent Line

$$\text{slope} = \frac{dy}{dx} = \frac{f(x0+h) - f(x0)}{h}$$

This slope is only an average or approximation, but what if we want the slope exactly at x0? We can just make h smaller and smaller to find this. We see this in Figure 4.51. We want to make it infinitely small (zero), so that we are dead on x0. Alas, there's a problem with this! Look at this limit:

$$\lim_{h \to 0} \frac{f(x0+h) - f(x0)}{h}$$

As *h* approaches zero, this limit looks like it might approach infinity, and we are fragged! However, this is the definition of the derivative:

Playrix Ex. 1023, page 372

07 0672318350 CH04 05•02•2003 2:36 PM Page 348

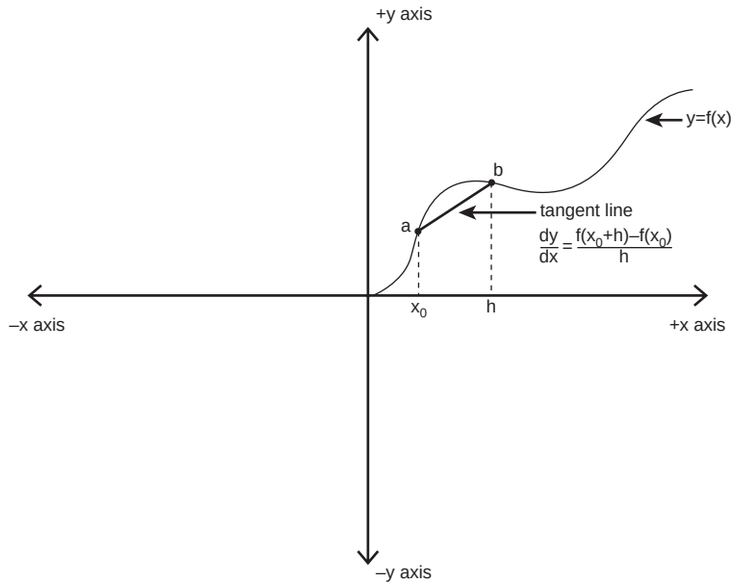


FIGURE 4.50 The definition of derivative.

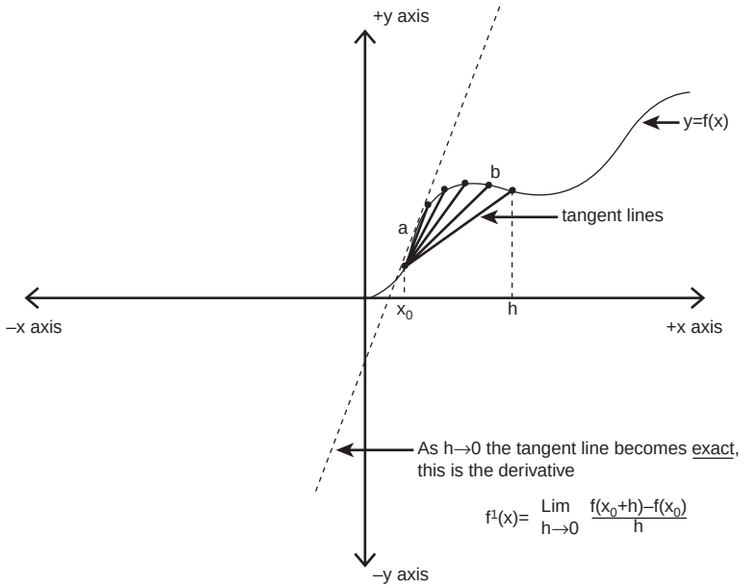


FIGURE 4.51 The tangent line becomes the derivative as h->0.

Playrix Ex. 1023, page 373

07 0672318350 CH04 05•02•2003 2:36 PM Page 349

EQUATION 4.41 Definition of the Derivative

$$f'(x) = \lim_{h \to 0} \frac{f(x0+h) - f(x0)}{h}$$

Just a sec, I know, I know. First, the new notation f'(x), or "f primed," means "the derivative of f with respect to x," so that's just notation. Similarly, f''(x) would mean "the second derivative of f with respect to x," or the derivative of the derivative. In general, these all mean the same thing:

1. f'(x)—The derivative of f with respect to x. Both the function and the variable are mentioned.
2. (d/dx)*f(x)—The derivative with respect to x of the function. Emphasizes that differentiation is an operator.
3. df/dx—Very similar to number 1, but in another format. Emphasizes both the function f and the variable to differentiate.

The important part about differentiation is the definition. If this limit exists, it is exactly the limit we need, and exactly the derivative. So how do we find the limit? Well, thankfully, a lot of really smart guys figured out hundreds (if not thousands) of general formulas to compute derivatives for us, so we don't have to (trust me, you don't want to use the definition!).

At this point, let's take a look at some of the more common mathematical expressions that we might have to differentiate in our domination of the universe and complete 3D simulation a la matrix techniques.

Derivatives of Simple Polynomials

Computing the derivatives of polynomials is the simplest, so let's start here. First, the derivative of the sum of terms in a polynomial is simply the sum of each of the derivatives:

$$f(x) = f_1(x) + f_2(x) + \ldots + f_n(x)$$
$$f'(x) = f_1'(x) + f_2'(x) + \ldots f_n'(x)$$

To compute the derivative of any polynomial term, here's what you do:

$$\text{Let } f(x) = a*x^n$$
$$f'(x) = n*a*x^{(n-1)}$$

Example 1:

$$f(x) = 5*x^3$$
$$f'(x) = 3*5*x^{(3-1)} = 15*x^2$$

Playrix Ex. 1023, page 374

07 0672318350 CH04 05•02•2003 2:36 PM Page 350

Example 2:

$f(x) = 3*x^2 + 2*x + 5$

or, writing out all the terms,

$$df/dx = 3*x^2 + 2*x^1 + 5*x^0$$
$$df/dx = 2*3*x^{(2-1)} + 1*2*x^{(1-1)} + 0*5*x^{(0-1)}$$
$$= 6*x^1 + 2*x^0 + 0$$
$$= 6*x + 2*1 = 6*x+2$$

NOTE

Note that the derivative of a constant is zero. Also, note that I switched notation for the fun of it, but it means the same thing. Many times, one notation is more convenient than another, as you will see.

Before moving on, let's look at a couple of general rules for computing derivatives.

Derivatives of Products: The Product Rule

Forget for a moment that you don't know how to differentiate everything, and just imagine that you have a product of two expressions. You can compute the derivative by thinking of each expression separately, rather than computing one big messy derivative. This is called the *product rule*. It looks like this:

Given u(x) and v(x) are functions of x,

$(d/dx)*(u * v) = u*(dv/dx) + v*(du/dx)$

which reads "The derivative of the product u multiplied by v is u multiplied by the derivative of v with respect to x added to v multiplied by the derivative of u with respect to x."

Example:

$f(x) = (3*x^2 + 3) * (5*x)$

Let

$u(x) = (3*x^2 + 3),\ v(x) = (5*x)$

then

$$f'(x) = u(x)*v'(x) + v(x)*u'(x)$$
$$= (3*x^2 + 3)*5 + (5*x)*(6*x)$$

Again, notice the change in notation, but it all means the same thing: d/dx, f'(x), df/dx.

Playrix Ex. 1023, page 375

07 0672318350 CH04 05•02•2003 2:36 PM Page 351

Derivatives of Quotients: The Quotient Rule

Computing the derivative of a quotient can be accomplished using the product rule and inverses, but to make life easier, we usually just use what's called the *quotient rule* and leave the problem in its original form. Here it is again, in a slightly modified notation:

Given u(x) and v(x) are functions of x,

$$\frac{d}{dx} * \frac{u}{v} = \frac{v\frac{du}{dx} - u\frac{dv}{dx}}{v^2}$$

which reads "The derivative of the quotient u divided by v is equal to v multiplied by the derivative of u with respect to x minus u multiplied by the derivative of v with respect to x, quantity divided by v squared."

Example:

Given

$$f(x) = \frac{4x^3 - 3}{-3x^2}$$

Let

$$u(x) = (4x^3 - 3),\ v(x) = (-3x^2)$$

then

$$f'(x) = \frac{[v(x)u'(x) - u(x)v'(x)]}{v(x)^2}$$

$$= \frac{[(-3x^2)(12x^2) - (4x^3 - 3)(-6x)]}{(-3x^2)^2}$$

Again, note the change of notation. Also, we could simplify this if we wanted to.

The Chain Rule

I was hoping to leave this out, but after I wrote a few more pages, I decided I couldn't. It's just so hard to write math notation with MS Word—I'm dying here! In any event, sometimes when computing derivatives, you might come upon functions that are compositions of functions. For example, take a look at the following function:

$$f(x) = \text{sqrt}(x^2+1) = (x^2+1)^{1/2}$$

Playrix Ex. 1023, page 376

07 0672318350 CH04 05•02•2003 2:36 PM Page 352

MATH

Taking the square root is the same as raising to the power of ?. I just can't make the square root symbol that easily!

We could attack this problem as it is and find the derivative of f(x) directly (if we can), or we could think of f(x) as a *composition* of two or more functions. *Composition* means nothing more than one function is the parameter to another. In this case, if we let g(x) = (x^2+1), and f(x) = sqrt(x), we can write this:

Given

$f(x) = sqrt(x) = (x)^{1/2}$

and

$g(x) = (x^2+1)$

If we replace the "x" in the definition of f(x) with the function for g(x), we have composed g with f, or mathematically:

$f(g(x)) = (f \bullet g)(x) = (x^2+1)^{1/2}$

NOTE

Note that the notation f(g(x)) and (f • g)(x) are equivalent.

As another example, let f(x) = sin x, and g(x) = (2*x+3), then f composed with g is

```
(f • g)(x) = sin (2*x+3)
```

The cool thing about thinking of functions as compositions is that it allows us to simplify them for a moment and work at a higher level. Now that you know what a composition is, let's get back to the chain rule for differentiation. The *chain rule* states that you can compute the derivative of a function by first computing the derivative of the composition, and then multiplying that by the derivative of the inside function. Then, if the inside function needs to be further simplified as more compositions, you continue the process. Mathematically, the chain rule can be stated as the following:

```
(d/dx)f(g(x)) = f'(g(x)) * g'(x)
```

Let's take another look at the first example with f(x) =$(x)^{1/2}$, g(x) = (x^2+1), and apply the rule.

$f(g(x)) = (x^2+1)^{1/2}$

Playrix Ex. 1023, page 377

07 0672318350 CH04 05•02•2003 2:36 PM Page 353

Then applying the chain rule, we have

```
f'(g(x)) = f'(g(x)) * g'(x)
     = (d/dx)(g(x)^1/2) * (d/dx) g(x)
     = ?*((x²+1)^-1/2 * (2*x)
   = x/(x²+1)^1/2
```

At this point, you can handle computing the derivative of just about anything you can think of. However, many times your functions might have trigonometric terms in them.

Derivatives of Trigonometric Functions

Showing the mathematical derivation for trigonometric functions is far beyond the scope of this book, so I am just going to give you a little table:

TABLE 4.7 Derivatives of Trigonometric Functions

Function	Derivative
sin x	cos x
cos x	-sin x
tan x	$\sec^2 x$
cot x	$-\csc^2 x$
sec x = 1/cos x	sec x * tan x
csc x = 1/sin x	-csc x * cot x

Simple enough. Unfortunately, most trig functions don't come in that pure of a flavor—they have functions of x as the parameter, and coefficients, and they might even be to the *n*th degree! However, these aren't that hard to handle. For example, suppose we have this:

```
f(x) = cos² x
```

Hmmmm. Let's try the product rule with

```
f(x) = cos(x) * cos(x)
```

Let

```
u = cos(x), let v = cos(x)
```

then

```
f'(x)   = u*(dv/dx) + v*(du/dx)
        = cos(x) * sin(x) + cos(x) * sin(x)
        = 2*cos(x) * sin(x)
```

Playrix Ex. 1023, page 378

07 0672318350 CH04 05•02•2003 2:36 PM Page 354

Cool, huh? The point is, you can usually figure out something, and of course there are billions of little plastic cheat sheets that have all kinds of trig derivatives. Let's look at a couple more. What about this

```
f(x) = a*transcendental(x)
```

meaning any constant *a* multiplied by a trig function. To compute the derivative, we can again use the product rule like this:

```
f(x) = a*transcendental(x)
```

Let u = a, v = transcendental(x), where the trig function could be anything. Then we have:

```
f'(x)   = u*(dv/dx) + (du/dx)*v
        = a * transcendental'(x) + 0
        = a * transcendental'(x)
```

Note that the derivative of u is 0, so the product of the second term is also 0, thus leaving us with the original constant multiplied by the derivative of the transcendental function. Did you see that coming? Here's an example:

Given

```
f(x) = 3*cos(x)
f'(x) = -3*sin(x)
```

Finally, let's look at problems of this form:

```
f(x) = a*cos(b*x+c), a*sin(b*x+c), a*tan(b*x+c), etc.
```

That is, there's a multiplier in the parameter part, along with a constant that models the time multiplier and a shift. This is just a composition, so we can use the chain rule here, letting the inside function be g(x) and the outside function be f(x). Let's try figuring out a*sin(b*x), as all the others are absolutely identical. The first thing to realize is that we can forget about the leading multiplier *a* because we can pull it out of the derivative anyway based on our previous results:

Given

```
f(x) = a*sin(b*x+c)
f'(x) = a*[(d/dx)sin(b*x+c)]
```

Thus, our problem really boils down to computing this:

```
f(x) = sin(b*x+c)
```

Playrix Ex. 1023, page 379

07 0672318350 CH04 05•02•2003 2:36 PM Page 355

Then we are done—we just multiply the answer by *a*. Let's use compositions and the chain rule for this baby. Here are the results:

Let

```
g(x) = (b*x+c)
f(x) = sin(x)
```

then

```
   f(g(x)) = sin (b*x+c)
f'(g(x))   = f'(g(x)) * g'(x)
           = (d/dx)sin(g(x)) * (d/dx)(b*x+c)
           = cos (b*x+c) * b
```

And now, multiply by the constant *a* to finish the problem (recall that we made the simplification f(x) = a*sin(b*x+c), f'(x) = a*[(d/dx)sin(b*x+c)]).

```
 f(x)   = a*sin(b*x)
f'(x)   = a*(cos(b*x+c)* b)
        = a*b*cos(b*x+c)
```

That's the quick tour of differentiation. At this point, you should be able to differentiate any polynomial or sum of polynomials that have transcendental terms that are themselves products of functions or compositions of functions. Hopefully, we won't need anything more complex than this!

Integrals

The next topic of discussion is integration. Integration is the opposite of differentiation—or *antiderivative*. If you were to differentiate f(x) to arrive at f'(x), what would you do to f'(x) to get back to the original f(x)? This process is called *integration*. Now, there are a lot of ways to mathematically define integration, along with ways to understand it geometrically or physically, so I'm going to give you several different points of view on the subject.

The first and purest way to look at the antiderivative is with its definition: Given a function f(x) and its derivative f'(x), the antiderivative F(x) is a function such that if you differentiate it, you get f(x):

```
F'(x) = f(x)
```

That seems fairly circular. It states that if you have a function f(x), and another function capital F(x) that you want to be the antiderivative of f(x), the derivative of F'(x) will equal the original function f(x). That's not that useful, so let's try an example like f(x) = 2*x. Now, the antiderivative of (2*x) is a function F(x), such that when I take the derivative of it, the result would be (2*x). Hmmmm. Let's write that down:

```
F'(x) = 2*x
```

Playrix Ex. 1023, page 380

07 0672318350 CH04 05•02•2003 2:36 PM Page 356

What function, when we take the derivative, equals (2*x)? The answer is x^2 because

$(d/dx)x^2 = 2*x^{(2-1)} = 2*x.$

So it looks like we have found it. If you're astute, you should realize that we have found a way to find the antiderivative of any term of the form $a*x^n$. We just reverse the process of differentiation, which states to multiply by the exponent and then decrease the exponent. Reversing that, we get the process for integration, which must be to divide by the exponent and then add one to the exponent. Mathematically, the antiderivative of $a*x^n$ is

$a*x^n = (a/(n+1))x^{*(n+1)} + c$

Don't freak about the "c" term. I will get to it shortly.

Hold onto that thought for a moment, and let's talk about integration as an operator, now that we know what we want it to do.

Definition of Antiderivative

Integration is the process of computing the antiderivative of a function, and is written using the ∫ symbol, like this:

$\int f(x)dx = F(x) + c$

f(x) is the function we want to find the antiderivative of, F(x) is the antiderivative, and c is the constant of integration. Let me show you what c represents. Let's say you have the function g(x) = 2*x+1 and you were to differentiate this:

$g(x) = 2*x^1+1$

$g'(x) = 1*2*x^{(1-1)} + 0$

$= 2$

The result is 2, which was just an example to show you that constants disappear when you differentiate. Now if we wanted to compute the antiderivative of that 2, knowing that we should end up back at (2*x+1), let's see what happens using our antiderivative rule relating functions of the form $a*x^n$ to the antiderivative $(a/(n+1))x^{*(n+1)}$:

$F'(x) = 2*x^0$

$F(x) = (2/1)*x(0+1) = 2*x$

Now you see there's a problem here. Basically, we took a function and found its derivative, and then used it as the new starting point and found its antiderivative thinking we would end up in the same place, but we didn't! We lost some information—the constant 1.0 in this case. This leads us back to the definition of an integral:

$\int f(x)dx = F(x) + c$

Playrix Ex. 1023, page 381

07 0672318350 CH04 05•02•2003 2:36 PM Page 357

The c takes this factor into consideration; that is, we know we could potentially lose information when we integrate to find the antiderivative. Only in cases where c = 0 can we forget about it; otherwise, we must keep this in mind. Thus in the example, we just computed that we need to add an arbitrary constant to the result, giving us (2*x+c). Hence, we really have a *family of functions*, or we have generated a function that is capable of generating all possible derivatives of the function 2. If this is confusing, don't worry about it too much. This is all theoretical, really; we just need to know how to integrate polynomials and transcendentals.

Let's recap now what we know. Integration is the opposite of differentiation, and the result is called the antiderivative. We know that the antiderivative of functions of the form $a*x^n$ is $[(a/(n+1))x^{*(n+1)} + c]$, where c is the constant of integration, and may be non-zero. In a moment we will talk about integrals of trig functions, but now I want to give you something more tangible to think about in relation to integration.

A Geometric Interpretation of Integration

As I said, differentiation is like computing the instantaneous rate of change or slope of a function. That is, when you differentiate a function f(x), you wind up with another function f'(x) that is the slope of f(x) for any x. Integration has a similar geometric interpretation in the opposite sense. Take a look at Figure 4.52. Here we see a common parabola function $f(x) = x^2$ in the interval of [0,5]. Additionally, I have shaded in the area under the curve. Interestingly enough, integration computes the area under the curve! Let's test that out. Given the initial function $f(x) = x^2$, let's integrate it using the formula:

$$\int f(x)dx = F(x) + c$$

and

$$f(x) = x^2$$
$$\int x^2dx = F(x) + c$$

Using our antiderivative formula $(a/(n+1))x^{*(n+1)}$, we get

$$\int x^2dx = x^3/3 + c$$

Now, that's great, but what's c? That's where the interval comes into play ([0,5]). We need this to lock into and compute c. The integrals we have been looking at thus far are called *indefinite integrals*—they compute a family of functions for the antiderivative because we don't know c, but definite integrals compute the *exact* value for c, and hence a single antiderivative. The only difference is that we write the definite integral a little differently.

Playrix Ex. 1023, page 382

07 0672318350 CH04 05•02•2003 2:36 PM Page 358

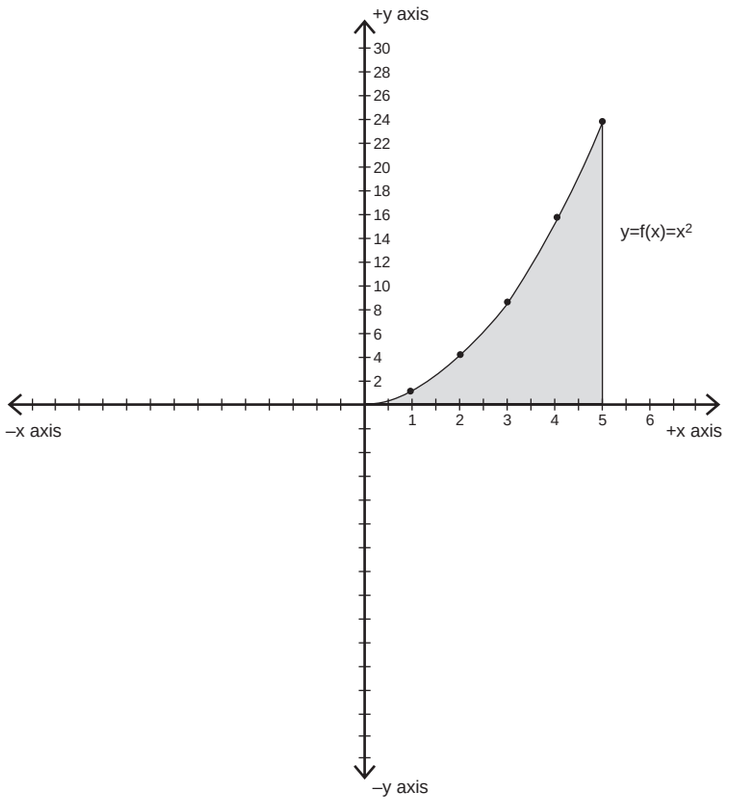


FIGURE 4.52 A geometric interpretation of integration.

EQUATION 4.42 The Definite Integral

$$\int_a^b f(x)dx$$

where [a,b] are the limits of integration. So, all you do is compute the integral as you normally would, but instead of just putting a generic "+c," we can compute it like this with the following notation and steps:

Step 1: Compute the integral.

$$\int_0^5 x^2dx = [x^3/3 + c]$$

Playrix Ex. 1023, page 383

07 0672318350 CH04 05•02•2003 2:36 PM Page 359

Step 2: Plug in the limits of integration and evaluate the integral at the upper limit and lower limit and subtract.

```
5
 = [x³/3] = [(5)³/3 - (0)³/3]
  0
 = [125/3 - 0]
 = 41.66
```

A lot is going on here, but basically you compute the integral as usual, then after you have the general form of the integral, you plug the upper limit of integration into the result to arrive at a number, then do the same with the lower limit, and then subtract. This gives you an exact answer.

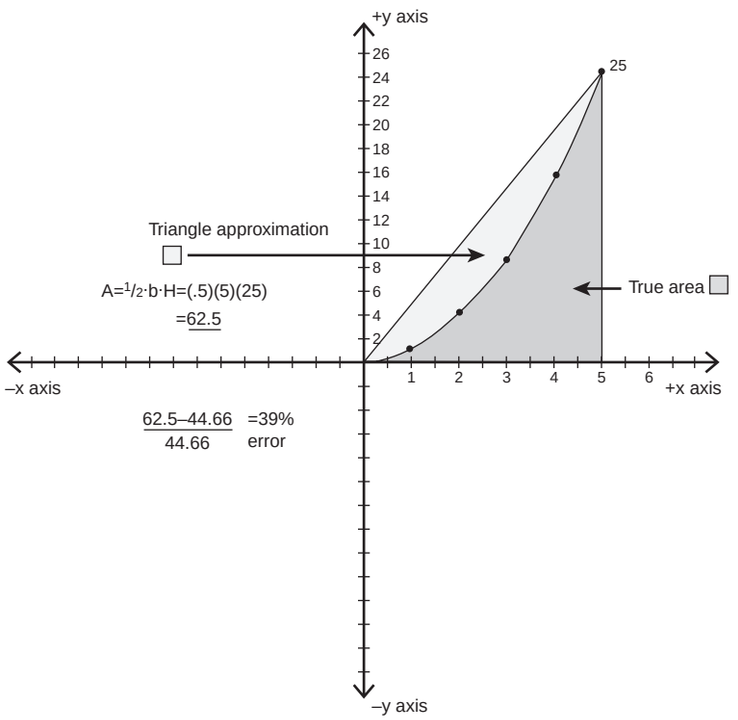


FIGURE 4.53 A triangular approximation of the area under the curve $f(x) = x^2$.

In this case, the answer should be the area under the curve $f(x) = x^2$ in the interval of [0,5]. We get 41.66 roughly; let's see whether that's in the ballpark. Because there are no formulas for computing areas under curves, we can use a triangular approximation, as shown in Figure 4.53. The height of the triangle is 25 and the width is 5, so the area is

```
A = ?*width*height = ?*5*25 = 62.5
```

Playrix Ex. 1023, page 384

07 0672318350 CH04 05•02•2003 2:36 PM Page 360

which tells us that we are in the ballpark, because the area under the curve is of course less. In fact, take my word for it—the area is exactly 41.66.

Integration Formulas

Okay, so now we have mathematical definition for integral, a geometric interpretation, and we know how to integrate both indefinite and definite integrals. Now let's take a look at some formulas for integration, as shown in Table 4.8.

TABLE 4.8 Integration Formulas

Integral	Antiderivative
$\int a*u^n\,du$	$(a/n+1)*u^{(n+1)} + c$
$\int \sin u\,du$	$-\cos u + c$
$\int \cos u\,du$	$\sin u + c$
$\int \sec^2 u\,du$	$\tan u + c$
$\int \csc^2 u\,du$	$-\cot u + c$
$\int \sec u * \tan u\,du$	$\sec u + c$
$\int \csc u * \cot u\,du$	$-\csc u + c$

Summing Up Integration with a Physics Interpretation

Actually, summing is exactly what integration does! It sums a function from one point to another. That's a very useful operation, and absolutely necessary when calculating physics models. Why? Because most of the time, you will have a model like this:

$$\text{Position: } x(t) = x_0 + v*t$$
$$\text{Velocity: } v(t) = v_0 + a*t$$

where the acceleration a is constant. The point of my example is that we know that the time derivative of the position x(t) is equal the velocity v(t), or (dx/dt)x(t) = v(t), or put yet another way:

$$x'(t) = v(t)$$

That means if I have the formula for v(t), its antiderivative must be equal to x(t). Mathematically, we have

$$x'(t) = v(t)$$

So, integrating both sides:

$$\int x'(t)\,dt = \int v(t)\,dt$$

The left-hand side is just x(t), because the antiderivative of the derivative of a function f'(x) must be f(x) again. Thus we have:

$$x(t) = \int v(t)\,dt$$

Playrix Ex. 1023, page 385

07 0672318350 CH04 05•02•2003 2:36 PM Page 361

And we already know the form of $v(t) = v_0+a*t$; plugging it into the indefinite integral, we get

$$x(t) = \int v_0+a*t\ dt = [v_0*t + {}^1/_2*a*t^2 + c]$$

And now all we need is c. But, if we use the initial conditions $t = 0$, $x = x_0$, and $v = v_0$, we get

$$[0 + 0 + c] = x_0$$

therefore $c = x_0$. Putting it all together, we get the final formula for motion as a function of time, and the initial position and velocity with constant acceleration:

$$x(t) = \int v_0+a*t\ dt = [v_0*t + {}^1/_2*a*t^2 + x_0]$$

The point of this is that we needed to integrate the velocity function to find the position! This is exactly what physics modeling is all about. You run the model, compute collisions and forces, and then you need to integrate numerically to find the new position of all the objects in the game. So yes, we need to know this stuff. When we get to physics modeling, I will show you how to integrate numerically instead of symbolically.

Summary

At this point, I don't know what to say—I feel like I have barely scratched the surface mathematically of what is needed in a video game these days. I never thought I would be writing a math book—a Kama Sutra, yes, but math?—no. However, we just don't have any more time or pages, so we have to move on. On the other hand, I think that if your math skills were indeed rusty (or nonexistent), you should feel pretty good at this point. You covered a lot of complex material: vectors, matrices, parametrics, complex numbers, quaternions, calculus, and more.

However, I strongly suggest you get a couple of good books on linear algebra, calculus, and physics. They can only help. This game stuff is just going to get harder and harder these days, as we are now breaching the barrier of "toy" and entering into "simulation." There is no doubt in my mind that in 3–5 years we will have computer games that run physics models that are absolutely realistic in the Newtonian scale of things. And that means math, math, math! So be warned: I'm not going to explain all the math behind everything from now on. I'll assume you know this level of math at least…

Playrix Ex. 1023, page 386

CHAPTER **5**

Building a Math Engine

"I memorized them..."

—Lazlo Hollyfeld, *Real Genius*

Well, I was hoping that the math engine and the last chapter would fit together into one chapter, but as usual there seems to be an exponentially growing number of pages needed to explain everything as each day passes. Writing a book on 3D is like trying to hit an F-22 Raptor by throwing a rock at it! You have to throw the rock so far ahead of it to hit the darn thing! Thus, instead of cutting things out, I decided to make the math library a separate chapter. Basically, everything covered in Chapter 4," It's a Math, Math, Math World—Trigonometry, Vectors, Matrices, and Quaternions" is going to be implemented in one way or another in code: vectors, matrices, quaternions, and so forth. This chapter is a code dump for the most part, because we've already covered how the math works. Thus, I won't show the internals of all the functions, just some of the representative ones. In most cases, the math functions are verbatim implementations of the mathematics from Chapter 4 converted to C/C++. Here's the layout of this chapter:

- Brief overview of the math engine
- Math constants
- Data structures
- Macros and inline functions
- Prototypes
- Globals
- Coordinate system support
- Vector support

IN THIS CHAPTER

- Brief Overview of the Math Engine
- Data Structures and Types
- Math Constants
- Macros and Inline Functions
- Prototypes
- Globals
- Math Engine API Listing
- Floating-Point Unit Math Primer
- Notes on Using the Math Engine
- Comments on Math Optimization

08 0672318350 CH05 05•02•2003 2:37 PM Page 364

- Matrix support
- 2D and 3D parametric line support
- 3D plane support
- Quaternion support
- Fixed-point math support
- Floating-point math primer

Brief Overview of the Math Engine

As I mentioned in the introduction paragraph, I was hoping to fit the math primer and code in a single chapter, but this stuff is just too complicated and I don't want to cut corners. Alas, this chapter is going to continue where the last chapter ended and we're going to implement all of our mathematical findings in a math engine. This engine will be the starting point for our 3D engine and we'll continually add to it, modify it, and improve it as need be. It's not the most efficient math library because it's not optimized too aggressively at the low level. That's so I can remember what the heck I did!

However, where possible, I've optimized it at the high level algorithmically. That is, for example, when computing the determinate of a matrix, I use co-factor expansion rather than the brute force method of computing determinates—this saves multiplications. However, in a real production 3D engine, there are many shortcuts that can be taken once you know what the engine needs to do during runtime. The point is, now we'll have some tools to work with, and we can worry about optimizing them later. Finally, 1GHz+ processors are common now, so it's not like I'm that worried!

File Structure of the Math Engine

The math engine is relatively simple as far as the files are concerned. There are only two new files:

`T3DLIB4.CPP`—The C++ source file

`T3DLIB4.H`—The header file

Figure 5.1 depicts the relationship of the new math engine to the rest of the files in our system. Notice that the math library `T3DLIB4.CPP` relies on some structures in `T3DLIB1.CPP`; thus, we always need to link to `T3DLIB1.CPP¦H` (which you would always do anyway). Moreover, `T3DLIB1.CPP¦H` relies on `DDRAW.H`, hence, that's `#included` in `T3DLIB4.CPP` also. The point is that if you want to use this library module as a standalone in something else, you'll need to move a couple of functions from `T3DLIB1.CPP`, and the `defines` from `T3DLIB1.H` into `T3DLIB4.*`, respectively—just a heads up.

Playrix Ex. 1023, page 389

08 0672318350 CH05 05•02•2003 2:37 PM Page 365

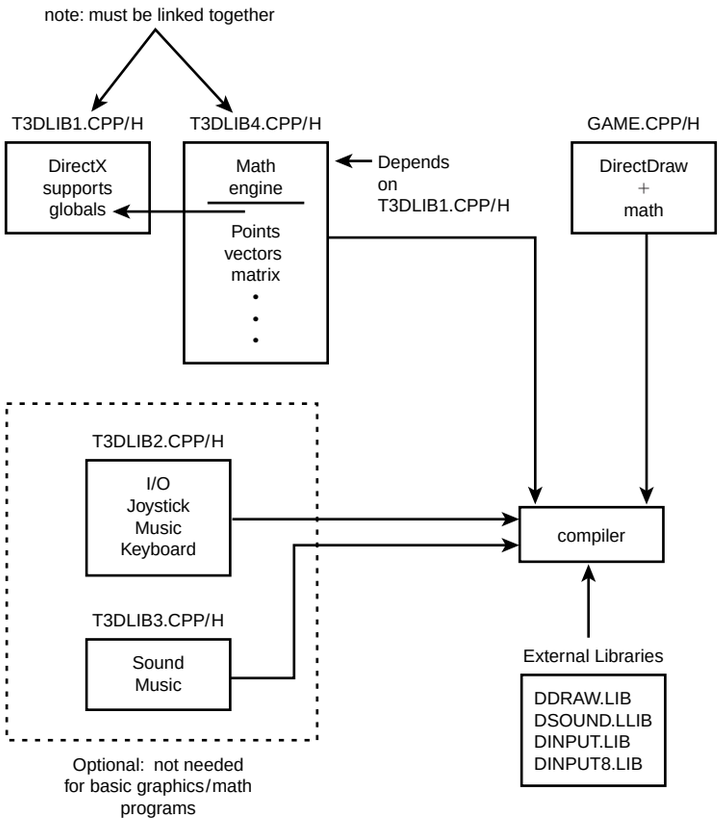


FIGURE 5.1 The relationship of the math engine to our game system.

The math engine itself is composed of numerous functions (and, believe me, it wasn't easy testing them all) that will help us work with points, vectors, lines, matrices, quaternions, and so on. Of course, I'm sure I've left stuff out, but we have a good foundation to work with.

Naming Conventions

As with the first *Tricks*, we'll use the convention that if we update a function, we'll append a number to the function name if the function performs the same operation, but only better or in a slightly different way. For example, a function named `Multiply()` might evolve into `Multiply2()` later in the book. Furthermore, naming functions is always one of those things everyone has a problem with because you want the function to have a reasonably intelligent name, but at the same time you don't want to type 80 billion characters! Therefore, I've decided on the convention that, in most cases, I'll use the

Playrix Ex. 1023, page 390

08 0672318350 CH05 05•02•2003 2:37 PM Page 366

class/structure name as part of the function name along with an almost-English description of what the function does and separate its data types by underscores (_). For example, the function that multiplies a vector and a matrix looks like this:

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va,
            MATRIX3X3_PTR mb,
            VECTOR3D_PTR vprod);
```

The function declaration pretty much sums up what the function does. OpenGL has a nice naming convention something like this, so what the heck? This is much better than something like this:

```
MMV3D33(…)
```

Although you might want to use a more aggressive naming convention similar to the above in a production engine, for us, that little extra typing will make things more clear.

NOTE

Additionally, classes and structures are always defined in all uppercase, as well as `#defined` macros. However, some inline macros (short functions) are in mixed case like standard functions.

Error Handling

Error handling is simple—there is none! This book is for intermediate/advanced programmers; if you want error handling and exceptions thrown, add them yourself. In most cases, each function checks for really stupid errors like divide by 0, but there are barely any return codes for any of the functions. If there is a return code, usually `1` (`TRUE`) means success and `0` (`FALSE`) means there was a problem.

My Final Word on C++

Throughout the years of developing games and game programming books, along with writing the previous volume to this book, I think that I've finally arrived at this fact: C++ is a terrible teaching language! It's great in production, and I use it for my own games, but as for teaching, it's very, very bad. I'm not going to get into why I think that, but it is. For example, universities still teach many programming courses and data structure courses in Pascal, Modula II, and Ada—why? Because they're very structured languages, and you know what each line of code does. C++ is too hard to figure out without prior knowledge of the classes defined by the author. Meaning that when someone is trying to learn an algorithm, why complicate matters with classes and constructors and overloaded operators?

Alas, the one library that I was really excited about making completely in C++ has found its final state in what I call C+ (C with some function overloading and a bit of C++). My

Playrix Ex. 1023, page 391

08 0672318350 CH05 05•02•2003 2:37 PM Page 367

reasoning is that my C code version is as fast or faster than the C++ version, and it's easier to understand. On the other hand, please convert the math library or anything else you like to C++ if it helps you.

Personally, it's the coolest thing being able to do matrix and vector math with overloaded operators, but it's too hard to know what's going on from a teaching point of view. Moreover, every time I want to update a C++ class-based math library, I have to write a new class and derive and overload the previous class—this is just too much work. I'd much rather have a `Multiply()` function and call the new one `Multiply2()`—capice? And that's all I have to say about that.

Data Structures and Types

Let's start by taking a look at the data structures and types the math engine uses. For the most part, 90% of the types and structures are new, but there are some that we created back in `T3DLIB1.CPP¦H` to do some simple matrix math. However, instead of breaking those up, I'm just going to make comments in the descriptions that they can be found in `T3DLIB1.H`.

The math engine supports numerous data types, including points, vectors, matrices, quaternions, parametric lines, 3D planes, polar coordinates, cylindrical coordinates, spherical coordinates, and fixed-point numbers. Let's take a look at each class of structures.

Vectors and Points

The math engine supports 2D, 3D, and 4D points and vectors, with 4D meaning homogenous coordinates of the form (x,y,z,w). However, in most cases, we'll never use w (because it will always be 1), but it's nice to have it. Anyway, because points and vectors are really the same thing—data-wise at least—the same data structure is used to hold them. Moreover, I've decided to use unions in the structures to allow multiple naming conventions. For example, sometimes it's nice to refer to a 3D point *p* as *p.x*, *p.y*, *p.z*, but other times, I want to use an algorithm that accesses x,y,z as an array *p.M[0], p.M[1], p.M[2]* or something like that. This is the theme throughout all the data structures. I want to give us a lot of room to refer to the data so that we can write clear algorithms. Hence, without further ado, here are the vector and point types:

```
// 2D vector, point without the w ////////////////////////
typedef struct VECTOR2D_TYP
{
union
  {
  float M[2]; // array indexed storage

  // explicit names
  struct
     {
```

Playrix Ex. 1023, page 392

08 0672318350 CH05 05•02•2003 2:37 PM Page 368


```
     float x,y;
     }; // end struct

  }; // end union

} VECTOR2D, POINT2D, *VECTOR2D_PTR, *POINT2D_PTR;

// 3D vector, point without the w ////////////////////////
typedef struct VECTOR3D_TYP
{
union
  {
  float M[3]; // array indexed storage

  // explicit names
  struct
     {
     float x,y,z;
     }; // end struct

  }; // end union

} VECTOR3D, POINT3D, *VECTOR3D_PTR, *POINT3D_PTR;

// 4D homogenous vector, point with w ////////////////////
typedef struct VECTOR4D_TYP
{
union
  {
  float M[4]; // array indexed storage

  // explicit names
  struct
     {
     float x,y,z,w;
     }; // end struct
  }; // end union

} VECTOR4D, POINT4D, *VECTOR4D_PTR, *POINT4D_PTR;
```

And here are a couple of old vertex defines from `T3DLIB1.H`:

```
// a 2D vertex
typedef struct VERTEX2DI_TYP
    {
```

Playrix Ex. 1023, page 393

08 0672318350 CH05 05•02•2003 2:37 PM Page 369


```
    int x,y; // the vertex
    } VERTEX2DI, *VERTEX2DI_PTR;

// a 2D vertex
typedef struct VERTEX2DF_TYP
    {
    float x,y; // the vertex
    } VERTEX2DF, *VERTEX2DF_PTR;
```

Later in the next chapter, when we start dealing with data structures to contain and represent 3D objects, we'll undoubtedly create 3D vertex representations.

Parametric Lines

In most cases, algorithms that use parametric representations of lines usually create them on the fly with points and variables that define the problem, but I thought that it might be nice to have an actual set of functions to work with explicit parametric lines. Hence, I created support for both 2D and 3D parametric lines. Here's the data structure for a 2D parametric line:


```
// 2D parametric line ////////////////////////////////////////
typedef struct PARMLINE2D_TYP
{
POINT2D p0; // start point of parametric line
POINT2D p1; // end point of parametric line
VECTOR2D v; // direction vector of line segment
       // ¦v¦=¦p0->p1¦
} PARMLINE2D, *PARMLINE2D_PTR;
```

As shown in Figure 5.2, **p0** represents the starting or initial point, **p1** represent the ending or terminal point, and **v** is the vector from **p0**->**p1**. If we get a little more ambitious (and we will), 3D parametric lines are just as easy to represent:

```
// 3D parametric line ////////////////////////////////////////
typedef struct PARMLINE3D_TYP
{
POINT3D p0; // start point of parametric line
POINT3D p1; // end point of parametric line
VECTOR3D v; // direction vector of line segment
       // ¦v¦=¦p0->p1¦
} PARMLINE3D, *PARMLINE3D_PTR;
```

Figure 5.3 represents a 3D parametric line. The values **p0**, **p1**, and **v** represent the same as with the 2D parametric line. However, you'll notice that I am using a left-handed system for the figure. Does this matter? No! A line is a line, no matter what the system. As long as you define the line and use the line in the same system, there won't be a problem.

Playrix Ex. 1023, page 394

08 0672318350 CH05 05•02•2003 2:37 PM Page 370

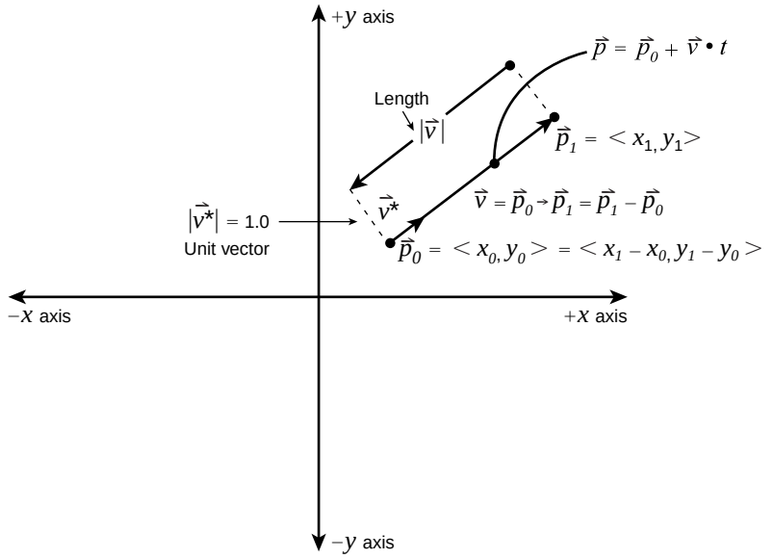


FIGURE 5.2 The 2D parametric line model.

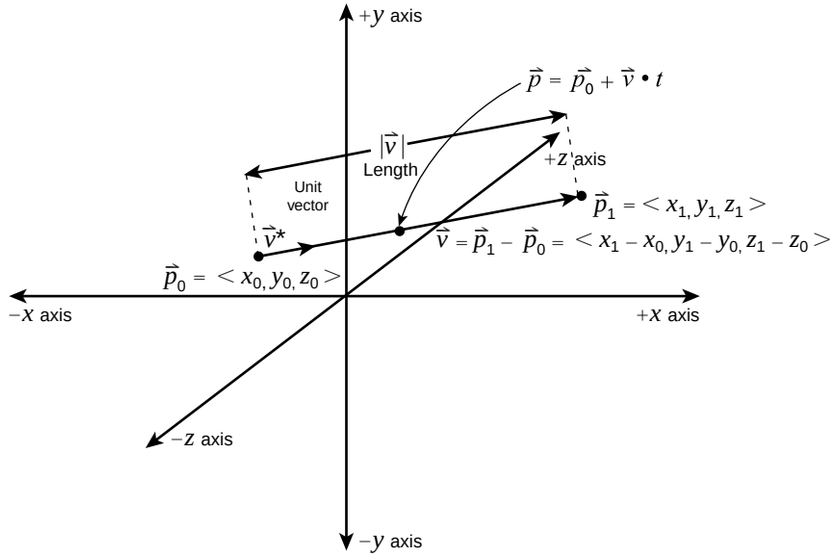


FIGURE 5.3 The 3D parametric line model.

Playrix Ex. 1023, page 395

08 0672318350 CH05 05•02•2003 2:37 PM Page 371

3D Planes

Although most of the time we worry about 3D planes, it will be in the context of polygons. I thought that it might be nice to have a 3D plane type for a few reasons: to be able to perform partitioning algorithms, to ease writing clipping algorithms, and finally to help with collision detection algorithms between parametric 3D lines and 3D planes. Anyway, there are a number of ways to represent a plane in 3D, but ultimately they all represent the same object. Thus, I used the *point-normal* form rather than the explicit form to hold the representation. That doesn't mean that we can't store a plane in the form

```
a*x+b*y+c*z+d = 0
```

It just means that we need to write a function that converts it into the point normal form for storage. Furthermore, the point normal form is more suited for a number of computations and not having to perform the conversion from explicit form is helpful. In any case, here's the data structure we'll use for now:

```
// 3D plane ////////////////////////////////////////////////////
typedef struct PLANE3D_TYP
{
POINT3D p0; // point on the plane
VECTOR3D n; // normal to the plane (not necessarily a unit vector)
} PLANE3D, *PLANE3D_PTR;
```

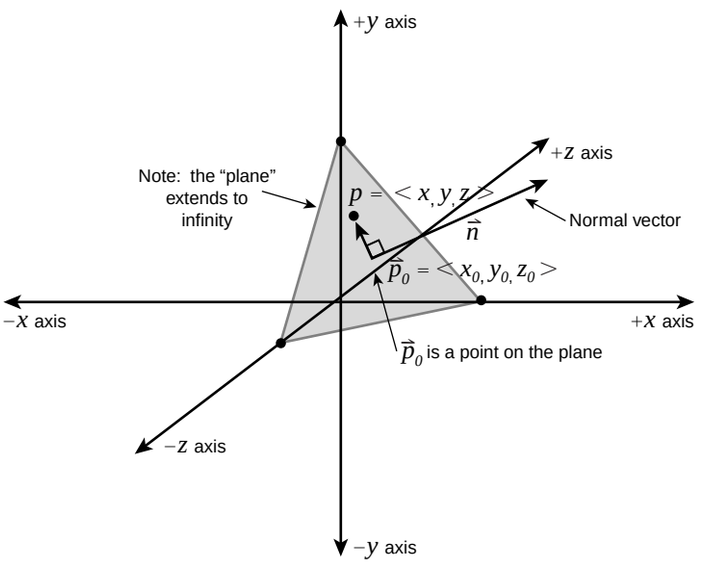



FIGURE 5.4 The 3D plane model.

Playrix Ex. 1023, page 396

08 0672318350 CH05 05•02•2003 2:37 PM Page 372

Figure 5.4 depicts a 3D plane (in an LHS), **p0** is a point on the plane, and the vector **n** is normal to the plane. Note: we don't assume that **n** is a unit vector.

Matrices

The largest set of data structures in the math engine is the matrices. There is support for 1×2, 2×2, 1×3, 3×2, 3×3, 1×4, 4×4, and 4×3. The 1×*n* matrices are really data equivalent to *n*-element vectors. For example, a 1×3 matrix has the same data footprint as a `VECTOR3D`, which is good because we can cast data types and get functions written for one type to work transparently with another. Anyway, here's all the matrix types. Note that many of the 2×2 and 3×3 are defined in `T3DLIB1.H` and all matrices are in row major form and indexed [0..n][0..n] = [row_index][column_index].

```
// matrix defines

// 4x4 matrix ////////////////////////////////////////////
typedef struct MATRIX4X4_TYP
{
union
  {
  float M[4][4]; // array indexed data storage

  // storage in row major form with explicit names
  struct
     {
     float M00, M01, M02, M03;
     float M10, M11, M12, M13;
     float M20, M21, M22, M23;
     float M30, M31, M32, M33;
     }; // end explicit names

  }; // end union

} MATRIX4X4, *MATRIX4X4_PTR;

// 4x3 matrix ////////////////////////////////////////////
typedef struct MATRIX4X3_TYP
{
union
  {
  float M[4][3]; // array indexed data storage

  // storage in row major form with explicit names
  struct
     {
```

Playrix Ex. 1023, page 397

08 0672318350 CH05 05•02•2003 2:37 PM Page 373


```
     float M00, M01, M02;
     float M10, M11, M12;
     float M20, M21, M22;
     float M30, M31, M32;
     }; // end explicit names

  }; // end union

} MATRIX4X3, *MATRIX4X3_PTR;

// 1x4 matrix ////////////////////////////////////////////
typedef struct MATRIX1X4_TYP
{
union
  {
  float M[4]; // array indexed data storage

  // storage in row major form with explicit names
  struct
     {
     float M00, M01, M02, M03;
     }; // end explicit names

   }; // end union
} MATRIX1X4, *MATRIX1X4_PTR;


// 3x3 matrix ////////////////////////////////////////////
// from T3DLIB1.H
typedef struct MATRIX3X3_TYP
    {
    union
    {
    float M[3][3]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01, M02;
       float M10, M11, M12;
       float M20, M21, M22;
       }; // end explicit names
```


Playrix Ex. 1023, page 398

08 0672318350 CH05 05•02•2003 2:37 PM Page 374


```
    }; // end union
    } MATRIX3X3, *MATRIX3X3_PTR;

// 1x3 matrix ////////////////////////////////////////////
// from T3DLIB1.H
typedef struct MATRIX1X3_TYP
    {
    union
    {
    float M[3]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01, M02;

       }; // end explicit names
    }; // end union
    } MATRIX1X3, *MATRIX1X3_PTR;

// 3x2 matrix ////////////////////////////////////////////
// from T3DLIB1.H
typedef struct MATRIX3X2_TYP
    {
    union
    {
    float M[3][2]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01;
       float M10, M11;
       float M20, M21;
       }; // end explicit names

    }; // end union
    } MATRIX3X2, *MATRIX3X2_PTR;


// 2x2 matrix ////////////////////////////////////////////
typedef struct MATRIX2X2_TYP
{
```

Playrix Ex. 1023, page 399

08 0672318350 CH05 05•02•2003 2:37 PM Page 375


```
union
  {
  float M[2][2]; // array indexed data storage

  // storage in row major form with explicit names
  struct
     {
     float M00, M01;
     float M10, M11;
     }; // end explicit names

   }; // end union
} MATRIX2X2, *MATRIX2X2_PTR;

// 1x2 matrix /////////////////////////////////////////////
// from T3DLIB1.H
typedef struct MATRIX1X2_TYP
    {
    union
    {
    float M[2]; // array indexed data storage

    // storage in row major form with explicit names
    struct
       {
       float M00, M01;

       }; // end explicit names
    }; // end union
    } MATRIX1X2, *MATRIX1X2_PTR;
```


Notice the naming conventions on all the matrices. For example, if we defined a 3×3 matrix like this:

```
MATRIX3X3 m = {0,1,2, 3,4,5, 6,7,8 };
```

Then the matrix would look like this in memory:

```
¦0   1   2¦
¦3   4   5¦
¦6   7   8¦
```

Playrix Ex. 1023, page 400

08 0672318350 CH05 05•02•2003 2:37 PM Page 376

which is in row major form. To access the lower-right hand element and assign a 100.0 to it, both of the following statements are equivalent because of the union:

```
m.M22 = 100; // all elements are 0 indexed

m.M[2][2] = 100;
```

Quaternions

The next data type is the 4D hyper complex quaternion (I just love saying that). If you recall, quaternions have the form

q = q0 + q1*i + q2*j + q3*k

or

q = q0 + <q1, q2, q3>

or

q = q0 + **qv**

Any way you look at it, a quaternion is nothing more than a 4-tuple, where the real part is q0, and the vector part is imaginary. Moreover, because we're going to use quaternions in the context of rotation and camera work, it will be advantageous to be able to represent them in the form of a single real q0 or w along with a vector part <x,y,z>. That way we can use some of the other VECTOR3D functions to do work on quaternions—possibly because the data footprints are similar? With that in mind, take a look at the definition of a quaternion:

```
// 4d quaternion /////////////////////////////////////////////
// note the union gives us a number of ways to work with
// the components of the quaternion
typedef struct QUAT_TYP
{
union
  {
  float M[4]; // array indexed storage w,x,y,z order

  // vector part, real part format
  struct
     {
     float  q0; // the real part
     VECTOR3D qv; // the imaginary part xi+yj+zk
     };
```

Playrix Ex. 1023, page 401

08 0672318350 CH05 05•02•2003 2:37 PM Page 377


```
  struct
     {
     float w,x,y,z;
     };
  }; // end union

} QUAT, *QUAT_PTR;
```

Notice the union allows access to the quaternion in three different ways: as an array, as a float vector pair, and explicitly as w,x,y,z, which is great depending on the algorithm the quaternion is being processed with. For example, here's a quaternion defined and then the real part is altered:

```
QUAT q = {1, 1,2,3 };

q.w  = 5; // access with explicit name
q.q0  = 5; // access with real float/vector name
q.M[0] = 5; // access with array
```


NOTE

It was tempting to put quaternions into the form <x,y,z,w> so that their data footprint matched that of a VECTOR4D. However, tempting as it was, I wanted to keep the convention of the data type in the same format as used when writing them mathematically.

TIP

Later, we might create a new quaternion type if we find that a different data ordering speeds calculations or access.

Angular Coordinate System Support

Although I doubt we'll use coordinate systems other than Cartesian throughout the book, there might be a time when we need it just to make a representation more natural to model the problem. For example, if we were modeling a weapon turret (as shown in Figure 5.5), a polar coordinate system might be perfect because we need an angular direction about the axis. Hence, the math engine supports all angular 2D- and 3D-coordinate systems and transformation between systems. The systems supported are *polar 2D*, *cylindrical 3D*, and *spherical 3D*. All angles are in radians, and all conversions are relative to a right-handed system, so watch out if you're using a left-handed system—you might get a negative sign if you don't take the right-handedness into consideration.

Playrix Ex. 1023, page 402

08 0672318350 CH05 05•02•2003 2:37 PM Page 378

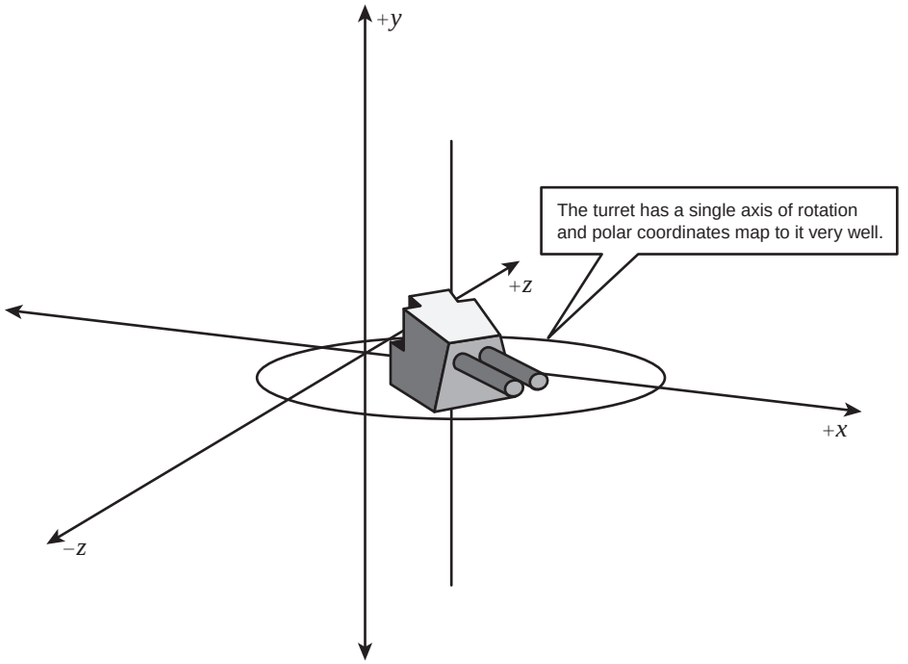


FIGURE 5.5 A weapon turret modeled with polar coordinates.

CAUTION

Watch out! All the angular functions assume that angles are in terms of radians NOT degrees!

Polar 2D

The polar 2D system is shown in Figure 5.6. A point is represented by a distance r from the origin or ***Pole***, and a heading or angle θ (theta). Thus, the notation **p**(r, θ) means that the point **p** is located an angle θ degrees (or radians) relative to a reference ray (usually the x-axis) measured counter-clockwise, and a distance r out on that directional heading. The data type is nothing more than this exact representation converted to C++:

```
// 2D polar coordinates ////////////////////////////////////////
typedef struct POLAR2D_TYP
{
float r;   // the radi of the point
float theta; // the angle in rads
} POLAR2D, *POLAR2D_PTR;
```

Playrix Ex. 1023, page 403

08 0672318350 CH05 05•02•2003 2:37 PM Page 379

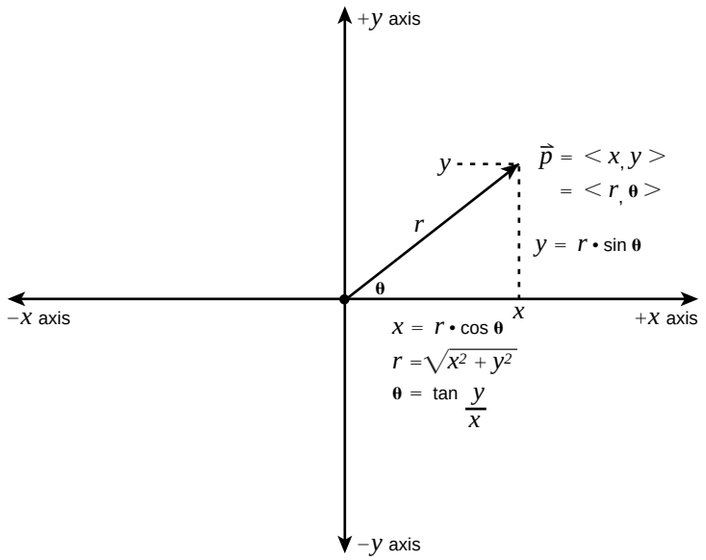


FIGURE 5.6 The polar 2D coordinate system.

Cylindrical 3D

The cylindrical 3D system is similar to the polar 2D system with the addition of a z-axis variable, as shown in Figure 5.7. Points are defined as a triple **p**(r,θ,z), meaning the point **p** is located an angle θ degrees (or radians) relative to a reference ray (usually the x-axis) measured counter-clockwise, and a distance r out on that directional heading. It's then raised or lowered an amount, z, on the z-axis. The data type contains exactly this information as shown in the following snippet:

```
// 3D cylindrical coordinates ////////////////////////////////
typedef struct CYLINDRICAL3D_TYP
{
float r;   // the radi of the point
float theta; // the angle in degrees about the z axis
float z;   // the z-height of the point
} CYLINDRICAL3D, *CYLINDRICAL3D_PTR;
```

Playrix Ex. 1023, page 404

08 0672318350 CH05 05•02•2003 2:37 PM Page 380

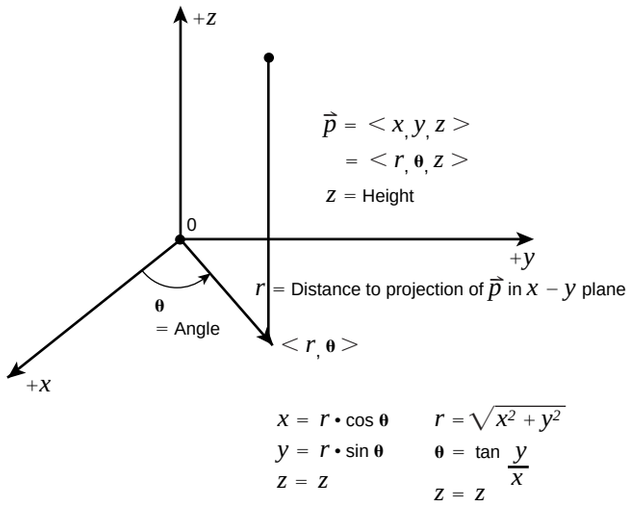


FIGURE 5.7 The cylindrical 3D coordinate system.

Spherical 3D

Spherical 3D coordinates are the most complex of all the angular systems. A point **p**(ρ,ϕ,θ) is defined by a distance and two angles as shown in Figure 5.8, where

ρ (rho) is the distance from the origin **o** to the point **p**.

ϕ (phi) is the angle that the directed line segment from the origin to **p** makes with the positive z-axis (thus, we'll use an RHS system in this analysis).

θ (theta) is the angle that projection of the line from **o** to **p** makes on the x-y plane, which is just the standard 2D polar θ. Also, 0 <= θ <= π; that is, θ must be in the interval from 0 to π, inclusive.

Of course, you've seen this before in the last chapter, but it can't hurt to show it again. Anyway, as you guessed the data structure is (drum roll please…):

```
// 3D spherical coordinates ////////////////////////////////////
typedef struct SPHERICAL3D_TYP
{
float p;   // rho, the distance to the point from the origin
float theta; // the angle from the z-axis and the line segment o->p
float phi;  // the angle from the projection if o->p onto the x-y
       // plane and the x-axis
} SPHERICAL3D, *SPHERICAL3D_PTR;
```

Playrix Ex. 1023, page 405

08 0672318350 CH05 05•02•2003 2:37 PM Page 381

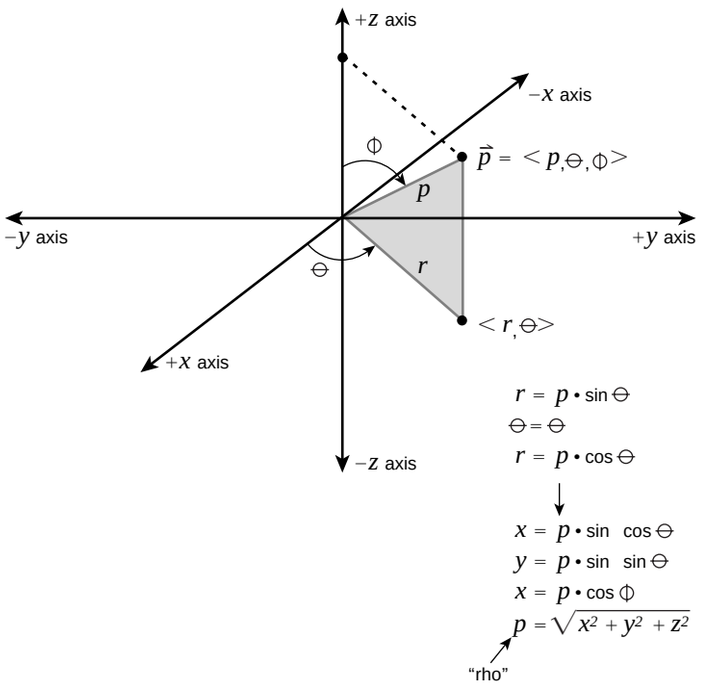


FIGURE 5.8 The 3D spherical coordinate system.

Fixed-Point Numbers

The last types are hardly types, but we need them nevertheless. Fixed-point numbers, as you know, are integers that are used to represent fractional or floating-point numbers with limited accuracy without the need of floating-point processing. Today the need for them is moot because Pentium+ and I64 processors can all perform floating-point math as quickly as fixed-point math. However, fixed-point sometimes still makes sense in tight inner loops where floating-point numbers can't be used, or when you want to use only integers, or on smaller handheld devices. In any case, for the majority of math in this book, we'll use the 16.16 format as shown in Figure 5.9; that is, 16 bits to represent the whole part and 16 bits to represent the decimal part. With that in mind, here are the data types:

```
// fixed point types ////////////////////////////////////////
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

Now, let's take a look the constants and defines used in the engine. Normally, they would come first, but I had some forward references to the data structures, so it was best to cover those first and the defines later.

Playrix Ex. 1023, page 406

08 0672318350 CH05 05•02•2003 2:37 PM Page 382

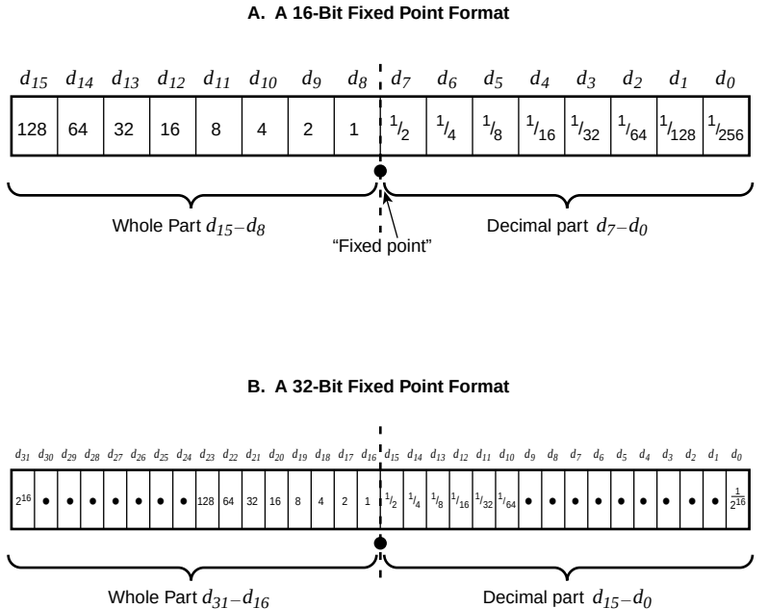


FIGURE 5.9 Fixed-point number representations.

Math Constants

The math engine is based on a number of defined constants. These constants are found in `T3DLIB4.H`, but there are a few legacy constants from `T3DLIB1.H` that are used. Here are the math constants that you'll find in `T3DLIB1.H` (just for review):

```
// math related constants from T3DLIB1.H

// pi defines
#define PI     ((float)3.141592654f)
#define PI2    ((float)6.283185307f)
#define PI_DIV_2  ((float)1.570796327f)
#define PI_DIV_4  ((float)0.785398163f)
#define PI_INV   ((float)0.318309886f)

// fixed point mathematics constants
#define FIXP16_SHIFT   16
#define FIXP16_MAG    65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

Playrix Ex. 1023, page 407

08 0672318350 CH05 05•02•2003 2:37 PM Page 383

Nothing more than pi and some fixed-point stuff. Note that we're using 16.16 fixed-point format, but more on that when we get to the fixed-point routines. Anyway, here are the constants from `T3DLIB4.H` broken down in sections with explanations where needed:

```
// defines for small numbers
#define EPSILON_E4 (float)(1E-4)
#define EPSILON_E5 (float)(1E-5)
#define EPSILON_E6 (float)(1E-6)
```

The *epsilon* constants help when performing math comparisons on small floating-point numbers. For example, it's common when doing floating-point math to test whether a number is equal to 0.0. However, 0.0 in floating-point is a rare thing once a few calculations have been performed and accuracy is lost, so a common strategy is to test for near 0.0 like this:

```
if (fabs(x) < 0.00001)
  {
  // that's close enough to 0
  } // end if
```


The next set of constants help with the parametric line functions and their return values (more on that when we get to the functions):

```
// defines for parametric line intersections
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

And now for some identity matrices to make initializing matrices easier:

```
// identity matrices

// 4x4 identity matrix
const MATRIX4X4 IMAT_4X4 = {1,0,0,0,
              0,1,0,0,
              0,0,1,0,
              0,0,0,1};

// 4x3 identity matrix (note this is not correct mathematically)
// but later we may use 4x3 matrices with the assumption that
// the last column is always [0 0 0 1]t
const MATRIX4X3 IMAT_4X3 = {1,0,0,
              0,1,0,
```

Playrix Ex. 1023, page 408

08 0672318350 CH05 05•02•2003 2:37 PM Page 384


```
              0,0,1,
              0,0,0,};


// 3x3 identity matrix
const MATRIX3X3 IMAT_3X3 = {1,0,0,
              0,1,0,
              0,0,1};

// 2x2 identity matrix
const MATRIX2X2 IMAT_2X2 = {1,0,
              0,1};
```

NOTE

Although only nxn matrices can have identity matrices (that is; 2x2, 3x3, 4x4, and so on), there's a 4x3 identity matrix in the preceding definitions. This technically isn't correct, but during many math operations, we use 4x3 matrices and assume the last column is $[0\ 0\ 0\ 1]^t$. Thus, we can create a 4x3 identity as long as we remember that there's an artificial extra column there when doing the math.

Amazingly enough, that's all there is to the math constants and defines!

Macros and Inline Functions

The problem with writing a math engine is that, in reality making function calls to perform operations on math objects can take as much time as the math operations themselves! Hence, using macros and inline functions is definitely the way to go. However, the problem with large inline functions is that inline functions **MUST** be available at compile time rather than at link time. That is, when you compile a program and call a function named `func()` in another module, the compiler doesn't need the code for `func()`, only the prototype:

```
int func(int x, int y);
```

With that, it can create a call in the compiled code and then later the linker can find `func()` and resolve the call to an address. The problem with inline functions is that during compile time the actual code **needs** to be available. Thus, the only way to cleanly implement large inline functions is with a header that defines the inline functions along with another pseudo C/C++ module named `*.inc` that you manually include in the module you're including. This way, the compiler has access to the actual code of the function you're requesting the inlining of. Figure 5.10 shows this setup.

Playrix Ex. 1023, page 409

08 0672318350 CH05 05•02•2003 2:37 PM Page 385

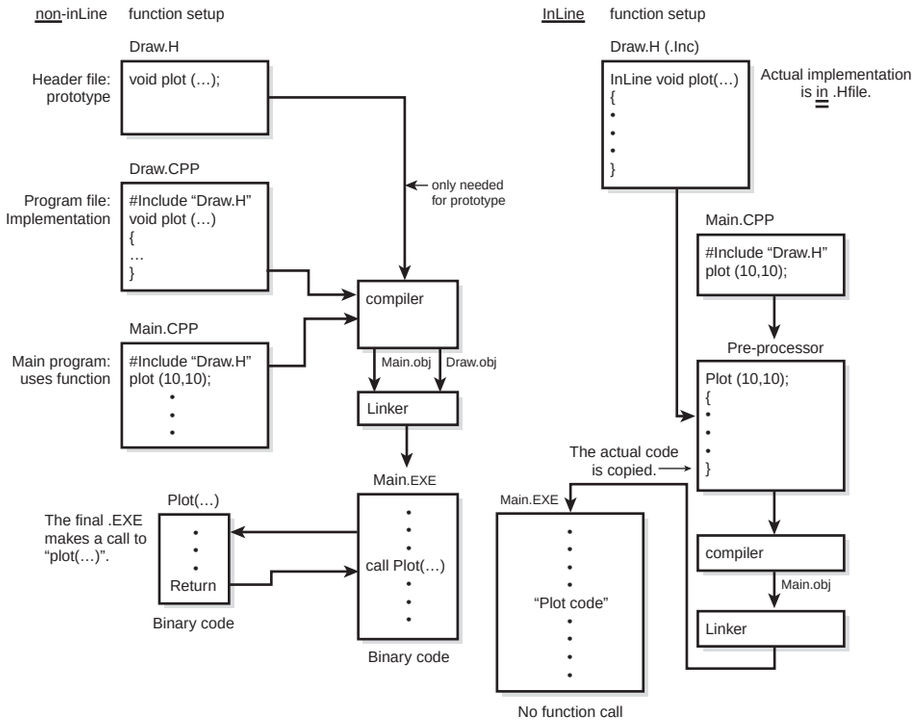


FIGURE 5.10 The inlining process in relation to the compiler and linker.

The point I am trying to make is this: When writing a math engine, all the functions should inline, for the most part, anything less than a few lines of code where the code itself takes on the same order of magnitude to run as does setting up the stack frame, making the call, and returning. For example, here's an example of a horrific function to call in an inner loop:

```
void Plot_Pixel(UCHAR *video_buffer, int mempitch, int x, int y, int color)
{
// plots a pixel
video_buffer[x+y*mempitch] = color;
} // end Plot_Pixel
```

If I make a call to this in my program, a function call will be generated. For example, take a look at the following code:

```
void main()
{
UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);
```

Playrix Ex. 1023, page 410

08 0672318350 CH05 05•02•2003 2:37 PM Page 386


```
for (int y=0; y < SCREEN_HEIGHT; y++)
     for (int x=0; x < SCREEN_WIDTH; x++)
        Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5); // slow call

} // end main
```

Although the function doesn't look bad, let's look at the disassembly of the code as generated by my Visual C++ 6.0 compiler:

```
_main PROC NEAR

; 52  : {

    push  ebp
    mov  ebp, esp
    sub  esp, 12                ; 0000000cH

; 53  : UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push  307200                ; 0004b000H
    call  _malloc
    add  esp, 4
    mov  DWORD PTR _video_buffer$[ebp], eax

; 54  :
; 55  : for (int y=0; y < SCREEN_HEIGHT; y++)

    mov  DWORD PTR _y$[ebp], 0
    jmp  SHORT $L43914
$L43915:
    mov  eax, DWORD PTR _y$[ebp]
    add  eax, 1
    mov  DWORD PTR _y$[ebp], eax
$L43914:
    cmp  DWORD PTR _y$[ebp], 480       ; 000001e0H
    jge  SHORT $L43916

; 56  :    for (int x=0; x < SCREEN_WIDTH; x++)

    mov  DWORD PTR _x$43917[ebp], 0
    jmp  SHORT $L43918
$L43919:
    mov  ecx, DWORD PTR _x$43917[ebp]
    add  ecx, 1
```

Playrix Ex. 1023, page 411

08 0672318350 CH05 05•02•2003 2:37 PM Page 387


```
    mov  DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp  DWORD PTR _x$43917[ebp], 640     ; 00000280H
    jge  SHORT $L43920

; 57  :         Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);

    push  5
    mov   edx, DWORD PTR _y$[ebp]
    push  edx
    mov   eax, DWORD PTR _x$43917[ebp]
    push  eax
    push  640                   ; 00000280H
    mov   ecx, DWORD PTR _video_buffer$[ebp]
    push  ecx
    call  ?Plot_Pixel@@YAXPAEHHHH@Z       ; Plot_Pixel
    add   esp, 20                ; 00000014H
    jmp   SHORT $L43919
$L43920:
    jmp   SHORT $L43915
$L43916:

; 58  :
; 59  : } // end main

    mov   esp, ebp
    pop   ebp
    ret   0
_main  ENDP
```

I've highlighted all the code needed to make the call to `Plot_Pixel()` to set up the stack frame, the call, and the fix up during the return. Now, compare that to the actual code for the `Plot_Pixel()` function itself:

```
?Plot_Pixel@@YAXPAEHHHH@Z PROC NEAR         ; Plot_Pixel

; 45  : {

    push  ebp
    mov  ebp, esp

; 46  : // plots a pixel
; 47  : video_buffer[x+y*mempitch] = color;
```

Playrix Ex. 1023, page 412

08 0672318350 CH05 05•02•2003 2:37 PM Page 388


```
    mov  eax, DWORD PTR _y$[ebp]
    imul  eax, DWORD PTR _mempitch$[ebp]
    mov  ecx, DWORD PTR _x$[ebp]
    add  ecx, eax
    mov  edx, DWORD PTR _video_buffer$[ebp]
    mov  al, BYTE PTR _color$[ebp]
    mov  BYTE PTR [edx+ecx], al

; 48  : } // end Plot_Pixel

    pop  ebp
    ret  0
?Plot_Pixel@@YAXPAEHHHH@Z ENDP          ; Plot_Pixel
```

As you can see, the call to `Plot_Pixel()` does almost as much work as the function does! However, if we inline the function, all this call overhead goes away. Here's the same example, but this time I tell the compiler to inline the function:

```
_main  PROC NEAR

; 52  : {

    push  ebp
    mov  ebp, esp
    sub  esp, 16                ; 00000010H

; 53  : UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push  307200                ; 0004b000H
    call  _malloc
    add  esp, 4
    mov  DWORD PTR _video_buffer$[ebp], eax

; 54  :
; 55  : for (int y=0; y < SCREEN_HEIGHT; y++)

    mov  DWORD PTR _y$[ebp], 0
    jmp  SHORT $L43914
$L43915:
    mov  eax, DWORD PTR _y$[ebp]
    add  eax, 1
    mov  DWORD PTR _y$[ebp], eax
$L43914:
```

Playrix Ex. 1023, page 413

08 0672318350 CH05 05•02•2003 2:37 PM Page 389


```
    cmp  DWORD PTR _y$[ebp], 480        ; 000001e0H
    jge  SHORT $L43916

; 56  :    for (int x=0; x < SCREEN_WIDTH; x++)

    mov  DWORD PTR _x$43917[ebp], 0
    jmp  SHORT $L43918
$L43919:
    mov  ecx, DWORD PTR _x$43917[ebp]
    add  ecx, 1
    mov  DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp  DWORD PTR _x$43917[ebp], 640     ; 00000280H
    jge  SHORT $L43920

; 57  :       Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);

    mov  DWORD PTR $T43941[ebp], 5
    mov  edx, DWORD PTR _y$[ebp]
    imul  edx, 640              ; 00000280H
    mov  eax, DWORD PTR _x$43917[ebp]
    add  eax, edx
    mov  ecx, DWORD PTR _video_buffer$[ebp]
    mov  dl, BYTE PTR $T43941[ebp]
    mov  BYTE PTR [ecx+eax], dl
    jmp  SHORT $L43919
$L43920:
    jmp  SHORT $L43915
$L43916:

; 58  :
; 59  : } // end main
```

This time, I've highlighted the inlined call to `Plot_Pixel()` and as you can see, it's much better! More code, but 200% faster on average.

I think I've shown that inlining is something that we want to do when the function being called has a small body, but it's a pain to implement because there's no good way to manage the files. I like .H and .CPP files, but I hate putting all my inline function listings along with code in the .H file, and I think adding a .INC file is asking for trouble. So, the moral to the story is this: Where possible, use `#define` macros as long as you don't foresee problems with the macro parameters being called multiple times.

Playrix Ex. 1023, page 414

08 0672318350 CH05 05•02•2003 2:37 PM Page 390

Also, it's okay to use short inline functions (macros) in the header files as long as they aren't too big because most programmers hunting for source code clues won't go into the .H files, but into the .CPP files. Additionally, by having large code bases in .H files, headers take longer to compile (of course, using precompiled headers solves that). Bearing that in mind, when you're done with your engine, you might want to convert many functions to inline that are short in nature. The problem is that you end up migrating all your .CPP files into .H files, and that's a management problem for your source.

I tried to find a common ground for all this drama and convert only simple functions to inline macros and keep them in the .H file. All the real functions went into the .CPP file. However, sometimes I probably should have decided to convert some of my real functions to inline, and some of my inlines to non-inline and put them in the .CPP file, but I think you get the point I'm trying to make. A math engine needs to be wickedly fast; thus, as much as possible of it should be inlined.

Now that I've bored our optimization experts who are reading this, let's move on to all the macros. They're once again classed into working groups for general utility: vectors, matrices, lines, planes, quaternions, and so on. Hence, I'll present them in these groups. And finally, the macros and inline functions have comments describing what each group does (it isn't like you're not going to figure out five lines of code, anyway) <BG>.

General Utility and Conversion

In general, the only generic macros are in `T3DLIB1.H`, so let's get those out of the way:

```
// used to compute the min and max of two expressions
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define MAX(a, b) (((a) > (b)) ? (b) : (a))

// used for swapping algorithm
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// some math macros
#define DEG_TO_RAD(ang) ((ang)*PI/180.0)
#define RAD_TO_DEG(rads) ((rads)*180.0/PI)

#define RAND_RANGE(x,y) ( (x) + (rand()%((y)-(x)+1)))
```

All of these are straightforward. I find the *`RAND_RANGE()`* macro, along with `MIN()` and `MAX()`, very useful.

Points and Vectors

As I mentioned earlier in the chapter, the data format of a point and vector in 2D, 3D, and 4D is identical. Hence, I'm going to group the point and vector macros together here.

Playrix Ex. 1023, page 415

08 0672318350 CH05 05•02•2003 2:37 PM Page 391

Also, some of the macros are written as inline functions to help with type checking. And many of them were too long to write as #defined macros.

```
// vector macros, note the 4D vector sets w=1
// vector zeroing macros
inline void VECTOR2D_ZERO(VECTOR2D_PTR v)
{(v)->x = (v)->y = 0.0;}

inline void VECTOR3D_ZERO(VECTOR3D_PTR v)
{(v)->x = (v)->y = (v)->z = 0.0;}

inline void VECTOR4D_ZERO(VECTOR4D_PTR v)
{(v)->x = (v)->y = (v)->z = 0.0; (v)->w = 1.0;}

// macros to initialize vectors with explicit components
inline void VECTOR2D_INITXY(VECTOR2D_PTR v, float x, float y)
{(v)->x = (x); (v)->y = (y);}

inline void VECTOR3D_INITXYZ(VECTOR3D_PTR v, float x, float y, float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z);}

inline void VECTOR4D_INITXYZ(VECTOR4D_PTR v, float x,float y,float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z); (v)->w = 1.0;}


// vector intialization macros to initialize with other vectors
inline void VECTOR2D_INIT(VECTOR2D_PTR vdst, VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void VECTOR3D_INIT(VECTOR3D_PTR vdst, VECTOR3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void VECTOR4D_INIT(VECTOR4D_PTR vdst, VECTOR4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }


// vector copying macros
inline void VECTOR2D_COPY(VECTOR2D_PTR vdst, VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void VECTOR3D_COPY(VECTOR3D_PTR vdst, VECTOR3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }
```


Playrix Ex. 1023, page 416

08 0672318350 CH05 05•02•2003 2:37 PM Page 392


```
inline void VECTOR4D_COPY(VECTOR4D_PTR vdst, VECTOR4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }


// point initialization macros
inline void POINT2D_INIT(POINT2D_PTR vdst, POINT2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_INIT(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void POINT4D_INIT(POINT4D_PTR vdst, POINT4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }

// point copying macros
inline void POINT2D_COPY(POINT2D_PTR vdst, POINT2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_COPY(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void POINT4D_COPY(POINT4D_PTR vdst, POINT4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }
```

As you can see, there's a macro for everything! Zeroing, initialization with different data types, and copying. Lots of redundancy, but who cares?

Matrices

Next are the matrix macros. Note that some are `#defined` macros, whereas others are short inline functions.

```
// matrix macros

// macros to clear out matrices
#define MAT_ZERO_2X2(m) {memset((void *)(m), 0, sizeof(MATRIX2X2));}
#define MAT_ZERO_3X3(m) {memset((void *)(m), 0, sizeof(MATRIX3X3));}
#define MAT_ZERO_4X4(m) {memset((void *)(m), 0, sizeof(MATRIX4X4));}
#define MAT_ZERO_4X3(m) {memset((void *)(m), 0, sizeof(MATRIX4X3));}
```

Playrix Ex. 1023, page 417

08 0672318350 CH05 05•02•2003 2:37 PM Page 393

NOTE

Many compilers use a byte-by-byte memory fill when calling the function `memset()`. We may be able to get more speed by using a quad-byte clear to zero out the matrices. Also note that, in general, integers do *NOT* equal their floating-point counterparts bit for bit for whole numbers; that is, the representation of (float)5.0 is completely different from (int)5. However, the 32-bit floating point representation of 0.0 is equal to the 32-bit integer representation of 0, thus we can use this trick and the memory functions to clear out floating-point arrays quickly.

```
// macros to set the identity matrix
#define MAT_IDENTITY_2X2(m) {memcpy((void *)(m), /
    (void *)&IMAT_2X2, sizeof(MATRIX2X2));}

#define MAT_IDENTITY_3X3(m) {memcpy((void *)(m), /
    (void *)&IMAT_3X3, sizeof(MATRIX3X3));}

#define MAT_IDENTITY_4X4(m) {memcpy((void *)(m), /
    (void *)&IMAT_4X4, sizeof(MATRIX4X4));}

#define MAT_IDENTITY_4X3(m) {memcpy((void *)(m), /
    (void *)&IMAT_4X3, sizeof(MATRIX4X3));}

// matrix copying macros
#define MAT_COPY_2X2(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
          (void *)(src_mat), sizeof(MATRIX2X2) ); }
#define MAT_COPY_3X3(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
          (void *)(src_mat), sizeof(MATRIX3X3) ); }
#define MAT_COPY_4X4(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
          (void *)(src_mat), sizeof(MATRIX4X4) ); }
#define MAT_COPY_4X3(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
           (void *)(src_mat), sizeof(MATRIX4X3) ); }

// matrix transposing macros
inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m)
{ MATRIX3X3 mt;
mt.M00 = m->M00; mt.M01 = m->M10; mt.M02 = m->M20;
mt.M10 = m->M01; mt.M11 = m->M11; mt.M12 = m->M21;
mt.M20 = m->M02; mt.M21 = m->M12; mt.M22 = m->M22;
memcpy((void *)m,(void *)&mt, sizeof(MATRIX3X3)); }

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m)
{ MATRIX4X4 mt;
mt.M00 = m->M00; mt.M01 = m->M10; mt.M02 = m->M20; mt.M03 = m->M30;
```


Playrix Ex. 1023, page 418

08 0672318350 CH05 05•02•2003 2:37 PM Page 394


```
mt.M10 = m->M01; mt.M11 = m->M11; mt.M12 = m->M21; mt.M13 = m->M31;
mt.M20 = m->M02; mt.M21 = m->M12; mt.M22 = m->M22; mt.M23 = m->M32;
mt.M30 = m->M03; mt.M31 = m->M13; mt.M32 = m->M22; mt.M33 = m->M33;
memcpy((void *)m,(void *)&mt, sizeof(MATRIX4X4)); }

inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10; mt->M02 = m->M20;
 mt->M10 = m->M01; mt->M11 = m->M11; mt->M12 = m->M21;
 mt->M20 = m->M02; mt->M21 = m->M12; mt->M22 = m->M22; }

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10; mt->M02 = m->M20; mt->M03 = m->M30;
 mt->M10 = m->M01; mt->M11 = m->M11; mt->M12 = m->M21; mt->M13 = m->M31;
 mt->M20 = m->M02; mt->M21 = m->M12; mt->M22 = m->M22; mt->M23 = m->M32;
 mt->M30 = m->M03; mt->M31 = m->M13; mt->M32 = m->M22; mt->M33 = m->M33;}

// small inline functions that could be re-written as macros, but would
// be less robust

// matrix and vector column swaping macros
inline void MAT_COLUMN_SWAP_2X2(MATRIX2X2_PTR m, int c, MATRIX1X2_PTR v)
{ m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1]; }

inline void MAT_COLUMN_SWAP_3X3(MATRIX3X3_PTR m, int c, MATRIX1X3_PTR v)
{ m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1]; m->M[2][c]=v->M[2]; }

inline void MAT_COLUMN_SWAP_4X4(MATRIX4X4_PTR m, int c, MATRIX1X4_PTR v)
{m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1];
 m->M[2][c]=v->M[2]; m->M[3][c]=v->M[3]; }

inline void MAT_COLUMN_SWAP_4X3(MATRIX4X3_PTR m, int c, MATRIX1X4_PTR v)
{m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1];
 m->M[2][c]=v->M[2]; m->M[3][c]=v->M[3]; }
```

Notice that all the matrix functions use pointers. This theme is used throughout the majority of functions in the math library. Sometimes it's nice to use the real objects, but many times the stack pushing/popping and copying eat up too many cycles.

Quaternions

The next set of macros has to do with quaternions. Pretty much all the quaternion functions (including the ones in the `T3DLIB4.CPP`) should be inline macros; that would have made the code more complicated, but keep it in mind. Here they are:

Playrix Ex. 1023, page 419

08 0672318350 CH05 05•02•2003 2:37 PM Page 395


```
// quaternion macros
inline void QUAT_ZERO(QUAT_PTR q)
{(q)->x = (q)->y = (q)->z = (q)->w = 0.0;}

inline void QUAT_INITWXYZ(QUAT_PTR q, float w, float x,float y,float z)
{ (q)->w = (w); (q)->x = (x); (q)->y = (y); (q)->z = (z); }

inline void QUAT_INIT_VECTOR3D(QUAT_PTR q, VECTOR3D_PTR v)
{ (q)->w = 0; (q)->x = (v->x); (q)->y = (v->y); (q)->z = (v->z); }

inline void QUAT_INIT(QUAT_PTR qdst, QUAT_PTR qsrc)
{(qdst)->w = (qsrc)->w; (qdst)->x = (qsrc)->x;
 (qdst)->y = (qsrc)->y; (qdst)->z = (qsrc)->z; }

inline void QUAT_COPY(QUAT_PTR qdst, QUAT_PTR qsrc)
{(qdst)->x = (qsrc)->x; (qdst)->y = (qsrc)->y;
(qdst)->z = (qsrc)->z; (qdst)->w = (qsrc)->w; }
```

Mostly they're macros to initialize a quaternion in whatever format you like.

Fixed-Point Math

Finally, here are the fixed-point math macros for conversion and extraction of fixed-point numbers. The rest of the fixed-point functions are written as real functions. They're definitely targets for inlining, but there's a lot of inline assembly language for the multiplication and division functions that I didn't want to hide in a header file.

```
// extract the whole part and decimal part from a fixed point 16.16
#define FIXP16_WP(fp) ((fp) >> FIXP16_SHIFT)
#define FIXP16_DP(fp) ((fp) && FIXP16_DP_MASK)

// convert integer and float to fixed point 16.16
#define INT_TO_FIXP16(i) ((i) << FIXP16_SHIFT)
#define FLOAT_TO_FIXP16(f) (((float)(f) * (float)FIXP16_MAG+0.5))

// convert fixed point to float
#define FIXP16_TO_FLOAT(fp) ( ((float)fp)/FIXP16_MAG)
```

Prototypes

At this point, let's take a look at all the function prototypes. Of course, you won't know what many of them do yet, but you should get an idea, and then when we do the function listing later, everything will become clear. Also, note that I will mark math functions

Playrix Ex. 1023, page 420

08 0672318350 CH05 05•02•2003 2:37 PM Page 396

from T3DLIB1.CPP¦H so that you can keep track of that. Again the functions are listed in groups for vectors, matrices, and so forth. Here they are:

```
// trig and general math functions
float Fast_Sin(float theta);
float Fast_Cos(float theta);

// distance functions (from T3DLIB1.CPP¦H)
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

// polar, cylindrical, spherical functions
void POLAR2D_To_POINT2D(POLAR2D_PTR polar, POINT2D_PTR rect);
void POLAR2D_To_RectXY(POLAR2D_PTR polar, float *x, float *y);
void POINT2D_To_POLAR2D(POINT2D_PTR rect, POLAR2D_PTR polar);
void POINT2D_To_PolarRTh(POINT2D_PTR rect, float *r, float *theta);
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl, POINT3D_PTR rect);
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl,
              float *x, float *y, float *z);
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect, CYLINDRICAL3D_PTR cyl);
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect,
               float *r, float *theta, float *z);
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph, POINT3D_PTR rect);
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph, float *x, float *y, float *z);
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect, SPHERICAL3D_PTR sph);
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect,
               float *p, float *theta, float *phi);

// 2d vector functions
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vsum);
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vdiff);
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va, VECTOR2D_PTR vscaled);
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);
void VECTOR2D_Build(VECTOR2D_PTR init, VECTOR2D_PTR term,
                    VECTOR2D_PTR result);
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);
```

Playrix Ex. 1023, page 421

08 0672318350 CH05 05•02•2003 2:37 PM Page 397


```
// 3d vector functions
void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vsum);
VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vdiff);
VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Scale(float k, VECTOR3D_PTR va);
void VECTOR3D_Scale(float k, VECTOR3D_PTR va, VECTOR3D_PTR vscaled);
float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Cross(VECTOR3D_PTR va,VECTOR3D_PTR vb,VECTOR3D_PTR vn);
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR3D_Length_Fast(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);
void VECTOR3D_Build(VECTOR3D_PTR init, VECTOR3D_PTR term,
                    VECTOR3D_PTR result);
float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Print(VECTOR3D_PTR va, char *name);

// 4d vector functions
void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vsum);
VECTOR4D VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vdiff);
VECTOR4D VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va, VECTOR4D_PTR vscaled);
float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Cross(VECTOR4D_PTR va,VECTOR4D_PTR vb,VECTOR4D_PTR vn);
VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
float VECTOR4D_Length(VECTOR4D_PTR va);
float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term,
          VECTOR4D_PTR result);
float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Print(VECTOR4D_PTR va, char *name);

// 2x2 matrix functions (note there others in T3DLIB1.CPP¦H)
void Mat_Init_2X2(MATRIX2X2_PTR ma,
         float m00, float m01, float m10, float m11);
void Print_Mat_2X2(MATRIX2X2_PTR ma, char *name);
float Mat_Det_2X2(MATRIX2X2_PTR m);
```


Playrix Ex. 1023, page 422

08 0672318350 CH05 05•02•2003 2:37 PM Page 398


```
void Mat_Add_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR msum);
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR mprod);
int Mat_Inverse_2X2(MATRIX2X2_PTR m, MATRIX2X2_PTR mi);
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X, MATRIX1X2_PTR B);

// 3x3 matrix functions

// from T3DLIB1.CPP¦H
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod);

// from T3DLIB1.CPP¦H
int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma,
          MATRIX3X3_PTR mb,
          MATRIX1X3_PTR mprod);

// from T3DLIB1.CPP¦H
int Mat_Mul_3X3(MATRIX3X3_PTR ma,
        MATRIX3X3_PTR mb,
        MATRIX3X3_PTR mprod);

// from T3DLIB1.CPP¦H
inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
            float m00, float m01,
            float m10, float m11,
            float m20, float m21);

void Mat_Add_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb, MATRIX3X3_PTR msum);
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb,
             VECTOR3D_PTR vprod);
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
void Mat_Init_3X3(MATRIX3X3_PTR ma,
            float m00, float m01, float m02,
            float m10, float m11, float m12,
            float m20, float m21, float m22);
void Print_Mat_3X3(MATRIX3X3_PTR ma, char *name);
float Mat_Det_3X3(MATRIX3X3_PTR m);
int Solve_3X3_System(MATRIX3X3_PTR A, MATRIX1X3_PTR X, MATRIX1X3_PTR B);

// 4x4 matrix functions
void Mat_Add_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR msum);
void Mat_Mul_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR mprod);
```

Playrix Ex. 1023, page 423

08 0672318350 CH05 05•02•2003 2:37 PM Page 399


```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma, MATRIX4X4_PTR mb,
          MATRIX1X4_PTR mprod);
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va, MATRIX4X4_PTR mb,
            VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR3D_4X3(VECTOR3D_PTR va, MATRIX4X3_PTR mb,
            VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR4D_4X4(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
            VECTOR4D_PTR vprod);
void Mat_Mul_VECTOR4D_4X3(VECTOR4D_PTR va, MATRIX4X4_PTR mb,
            VECTOR4D_PTR vprod);
int Mat_Inverse_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mi);
void Mat_Init_4X4(MATRIX4X4_PTR ma,
           float m00, float m01, float m02, float m03,
           float m10, float m11, float m12, float m13,
           float m20, float m21, float m22, float m23,
           float m30, float m31, float m32, float m33);
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);

// quaternion functions
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
void QUAT_Scale(QUAT_PTR q, float scale);
float QUAT_Norm(QUAT_PTR q);
float QUAT_Norm2(QUAT_PTR q);
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
void QUAT_Normalize(QUAT_PTR q);
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Unit_Inverse(QUAT_PTR q);
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Inverse(QUAT_PTR q);
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR q3,
            QUAT_PTR qprod);
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta);
void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v, float theta);
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float theta_y,
          float theta_x);
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta);
void QUAT_Print(QUAT_PTR q, char *name);

// 2d parametric line functions
void Init_Parm_Line2D(POINT2D_PTR p_init, POINT2D_PTR p_term,
```


Playrix Ex. 1023, page 424

08 0672318350 CH05 05•02•2003 2:37 PM Page 400


```
           PARMLINE2D_PTR p);
void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t, POINT2D_PTR pt);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
              float *t1, float *t2);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
              POINT2D_PTR pt);

// 3d parametric line functions
void Init_Parm_Line3D(POINT3D_PTR p_init, POINT3D_PTR p_term,
           PARMLINE3D_PTR p);
void Compute_Parm_Line3D(PARMLINE3D_PTR p, float t, POINT3D_PTR pt);

// 3d plane functions
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
         VECTOR3D_PTR normal, int normalize);
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane);
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline, PLANE3D_PTR plane,
                 float *t, POINT3D_PTR pt);

// fixed point functions
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2);
void FIXP16_Print(FIXP16 fp);
```

Lots of functions, huh? Like I said, this math engine has support for just about everything we'll need to write a 3D game, even with this early version. And if we need optimization, we'll optimize, but right now, we have a solid working set of math functions. I can't wait to show you how they work. Any volunteers?

Globals

There aren't that many globals in the math engine. In fact, the only globals that exist are from the original minimal math support in `T3DLIB1.CPP¦H`. The globals were the sine and cosine lookup tables, as shown here:

```
// storage for our lookup tables from T3DLIB1.CPP¦H
extern float cos_look[361]; // 1 extra so we can store 0-360 inclusive
extern float sin_look[361]; // 1 extra so we can store 0-360 inclusive
```

Notice that these tables are in degrees not radians, and there are 361 entries that represent [0..360] degrees, inclusive. Now, you might say that 360 degrees is exactly the same as 0 degrees. And yes, it is, but having that last entry makes some of the algorithms easier to write and saves a single compare. The only thing you have to do to initialize the tables is call `Build_Sin_Cos_Tables()` at the beginning of your application.

Playrix Ex. 1023, page 425

08 0672318350 CH05 05•02•2003 2:37 PM Page 401

MATH

In a more robust math engine, you might have a number of global state variables that track states of the math engine, or more complex data structures such as matrix stacks (like those used in APIs such as D3D and OpenGL).

Math Engine API Listing

Now for the fun part! What we're going to do now is take a look at the entire math engine API from `T3DLIB4.CPP¦H` and see what each function does along, with a simple example and demo program for some of the more complex math API functions. Additionally, I'll show you the inside of many of the functions that are representative of other similar functions so that you can see how everything is written. Finally, for functions that aren't part of `T3DLIB4.CPP¦H` but are part of `T3DLIB1.CPP¦H`'s math support, descriptions can be found in Chapter 3, "3D Game Programming with a Virtual Computer."

COOL

The Texas Instruments TI-89 calculator has got to be the coolest thing on earth (besides the TI-92 and the newer models, of course). But these next-generation calculators with 3D graphing, symbolic algebra, matrix mathematics, and calculus support are a must to check out your math. Unless you're a mathematical sadist and like doing page after page of multiplication and division—yuck!

Trigonometric Functions

Function Prototype(s):

```
float Fast_Sin(float theta);

float Fast_Cos(float theta);
```

Function Listing Reference:

```
float Fast_Sin(float theta)
{
// this function uses the sin_look[] lookup table, but
// has logic to handle negative angles as well as fractional
// angles via interpolation, use this for a more robust
// sin computation that the blind lookup, but with with
// a slight hit in speed

// convert angle to 0-359
theta = fmodf(theta,360);
```

Playrix Ex. 1023, page 426

08 0672318350 CH05 05•02•2003 2:37 PM Page 402


```
// make angle positive
if (theta < 0) theta+=360.0;

// compute floor of theta and fractional part to interpolate
int theta_int  = (int)theta;
float theta_frac = theta - theta_int;

// now compute the value of sin(angle) using the lookup tables
// and interpolating the fractional part, note that if theta_int
// is equal to 359 then theta_int+1=360, but this is fine since the
// table was made with the entries 0-360 inclusive
return(sin_look[theta_int] +
    theta_frac*(sin_look[theta_int+1] - sin_look[theta_int]));

} // end Fast_Sin
```

Purpose:

`float Fast_Cos()` compute the sine and cosine of the sent parameter by using the `sin_look[]` and `cos_look[]` lookup table along with linear interpolation to arrive at a more accurate value than using the lookup table alone, while at the same time being faster than using the intrinsic C/C++ math library's `sin()` and `cos()` functions. The theta parameter must be in degrees. Figure 5.11 shows an illustration of the how the functions interpolate the value from the table.

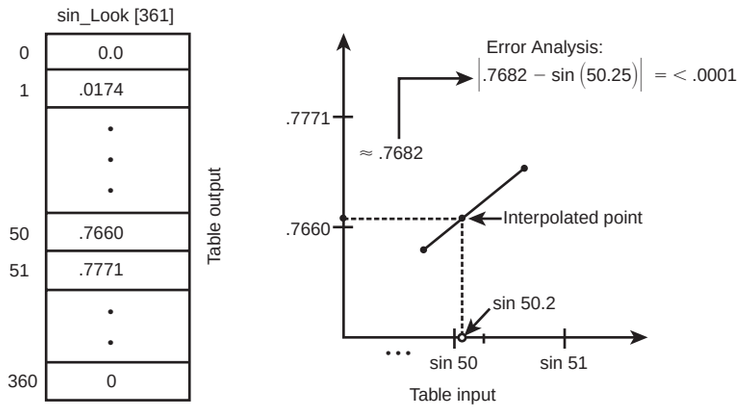


FIGURE 5.11 Interpolation to increase accuracy of trigonometric lookup tables.

Playrix Ex. 1023, page 427

08 0672318350 CH05 05•02•2003 2:37 PM Page 403

Example:

As an example, here's how you would compute the sine and cosine values of 50.2 degrees.

```
// build look up tables first
Build_Sin_Cos_Tables();

float answer1 = Fast_Sin(50.2)
float answer2 = Fast_Cos(50.2)
```

Coordinate System Support Functions

The next section of the API deals with conversions to and from Cartesian coordinates—polar, cylindrical, and spherical. Each of the functions is implemented identically to the math from Chapter 4, so there's no need to look at the innards of any of them unless you want to see statements like

```
"x=r*sin(theta);"
```

be converted to C/C++:

```
"x=r*sin(theta);".
```


Also, all the functions have a return type void. So, without further ado, let's take a look at the functions.

Function Prototype:

```
void POLAR2D_To_POINT2D(POLAR2D_PTR polar, POINT2D_PTR rect);
```

Purpose:

`void POLAR2D_To_POINT2D()` converts a 2D polar point in terms of r and theta into an (x,y) point and stores it in a `POINT2D` structure.

Example:

```
POLAR2D pp = {5, PI2};
POINT2D pr = {0,0};

// convert polar to rectangular
POLAR2D_To_POINT2D(&pp, &pr);
```

Function Prototype:

```
void POLAR2D_To_RectXY(POLAR2D_PTR polar, float *x, float *y);
```

Playrix Ex. 1023, page 428

08 0672318350 CH05 05•02•2003 2:37 PM Page 404

Purpose:

`void POLAR2D_To_RectXY()` converts a polar 2D point into explicit x,y coordinates.

Example:

```
POLAR2D pp = {5, PI2};
float x=0, y=0;

// convert polar point to x,y
POLAR2D_To_RectXY(&pp, &x, &y)
```

Function Prototype:

```
void POINT2D_To_POLAR2D(POINT2D_PTR rect, POLAR2D_PTR polar);
```

Purpose:

`void POINT2D_To_POLAR2D()` converts a point in rectangular format to 2D polar format.

Example:

```
POLAR2D pp = {0, 0};
POINT2D pr = {10,20};

// convert rectangular to polar
POINT2D_To_POLAR2D(&pr, &pp);
```

Function Prototype:

```
void POINT2D_To_PolarRTh(POINT2D_PTR rect, float *r, float *theta);
```

Purpose:

`void POINT2D_To_PolarRTh()` converts a polar 2D point into an explicit r and theta.

Example:

```
POLAR2D pp = {3, PI};
float r =0, theta = 0;

// convert polar to r, theta
POINT2D_To_PolarRTh(&pp, &r, &theta);
```

Function Prototype:

```
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl, POINT3D_PTR rect);
```

Playrix Ex. 1023, page 429

08 0672318350 CH05 05•02•2003 2:37 PM Page 405

Purpose:

`void CYLINDRICAL3D_To_POINT3D()` converts a cylindrical 3D point into a rectangular 3D point.

Example:

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r, theta, z
POINT3D pr = {0,0,0};

// convert cylindrical to rectangular
CYLINDRICAL3D_To_POINT3D(&pc, &pr);
```

Function Prototype:

```
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl, float *x, float *y, float *z);
```

Purpose:

`void CYLINDRICAL3D_To_RectXYZ()` converts a point from cylindrical 3D coordinates to explicit x,y,z coordinates.

Example:

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r, theta, z
float x=0, y=0, z=0;

// convert cylindrical to explicit x,y,z
CYLINDRICAL3D_To_RectXYZ(&pc, &x, &y, &z);
```

Function Prototype:

```
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect, CYLINDRICAL3D_PTR cyl);
```

Purpose:

`void POINT3D_To_CYLINDRICAL3D()` converts a rectangular 3D point into a cylindrical 3D point.

Example:

```
CYLINDRICAL3D pc = {0,0,0}; // r, theta, z
POINT3D pr = {1,2,3};

// convert rectangular to cylindrical
POINT3D_To_CYLINDRICAL3D(&pr, &pc);
```

Playrix Ex. 1023, page 430

08 0672318350 CH05 05•02•2003 2:37 PM Page 406

Function Prototype:

```
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect, float *r,
               float *theta, float *z);
```

Purpose:

`void POINT3D_To_CylindricalRThZ()` converts a 3D point into explicit cylindrical coordinates r, theta, z.

Example:

```
POINT3D pr = {1,2,3};
float r=0, theta=0, z=0;

// convert 3D point into explicit cylindrical coords
POINT3D_To_CylindricalRThZ(&pr, &r, &theta, &z);
```

Function Prototype:

```
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph, POINT3D_PTR rect);
```

Purpose:

`void SPHERICAL3D_To_POINT3D()` converts a spherical 3D point into a rectangular 3D point.

Example:

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
POINT3D pr = {0,0,0};

// convert from spherical to rectangular
SPHERICAL3D_To_POINT3D(&ps, &pr);
```

Function Prototype:

```
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph,
             float *x, float *y, float *z);
```

Purpose:

`void SPHERICAL3D_To_RectXYZ()` converts a spherical 3D point into explicit x,y,z coordinates.

Example:

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
float x=0, y=0, z=0;

// convert from spherical to rectangular
SPHERICAL3D_To_RectXYZ(&ps, &x, &y, &z);
```

Playrix Ex. 1023, page 431

08 0672318350 CH05 05•02•2003 2:37 PM Page 407

Function Prototype:

```
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect, SPHERICAL3D_PTR sph);
```

Purpose:

`void POINT3D_To_SPHERICAL3D()` converts a 3D point into spherical coordinates.

Example:

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p, theta, phi
POINT3D pr = {0,0,0};

// convert from rectangular to spherical
POINT3D_To_SPHERICAL3D(&pr, &ps);
```

Function Prototype:

```
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect, float *p,
               float *theta, float *phi);
```

Purpose:

`void POINT3D_To_SphericalPThPh()` converts a 3D point into explicit spherical coordinates rho, theta, phi.

Example:

```
POINT3D pr = {10,20,30};
float p=0, theta=0, phi=0;

// convert the 3D point to spherical coords
POINT3D_To_SphericalPThPh(&pr, &p, &theta, &phi);
```

Vector Support Functions

The vector support for the math engine comes in three flavors: 2D, 3D, and 4D. The 2D and 3D support are easy enough to understand, but the 4D is worth a quickie. Remember homogenous coordinates? For example, **p** = <x,y,z,w>, and if we want to convert **p** into a real 3D coordinate **p**' then we perform the following transform:

```
p' = p/w = <x/w, y/w, z/w, w/w>
```

And then we typically throw away the last component, giving

```
p' = <x/w, y/w, z/w>
```

Playrix Ex. 1023, page 432

08 0672318350 CH05 05•02•2003 2:37 PM Page 408

MATH

Note that many operations on 4D vectors treat w as a dummy, and work with only the first <x,y,z> components.

In any case, that's our refresher course on 4D homogenous coordinates. With that in mind, there are vector functions for all operations on 2D, 3D, and 4D vectors. Of course, some operations don't really make sense on each respective system. For example, taking the cross product of 2D vectors isn't defined. But, for the most part, there is a 2D, 3D, and 4D version of all vector functions, so I'm only going to show examples of one system. Moreover, vector functions that make sense only for a certain dimension will be illustrated separately, and vector functions that are similar will be grouped. Additionally, most of the functions return values that are either floats or the vector type of the function; for example, there are vector adding functions for 2D, 3D, and 4D vectors, and each returns a 2D, 3D, and 4D vector, respectively. Also, when referring to 2D, 3D, and 4D, I will use the notation "*D" to mean all of them. Get it? If not, you will!

NOTE

The data types for points and vectors are the same. Therefore, anywhere you use a `VECTOR3D` (for example), you can use a `POINT3D`. Of course, performing certain calculations on points doesn't really make sense. For example taking the cross product of two points. However, when building a vector, it makes sense to use a pair of points or a pair of vectors, or a point and a vector, and so forth.

Function Prototype(s):

```
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vsum);

void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vsum);

void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vsum);
```

Purpose:

`void VECTOR*D_Add()` adds the sent vectors (`va+vb`) as shown in Figure 5.12, and stores the results in `vsum`.

Example:

```
VECTOR3D v1 = {1,2,3};
VECTOR3D v2 = {5,6,7};
VECTOR3D vsum; // holds result

// add em
VECTOR3D_Add(&v1, &v2, &vsum);
```

Playrix Ex. 1023, page 433

08 0672318350 CH05 05•02•2003 2:37 PM Page 409

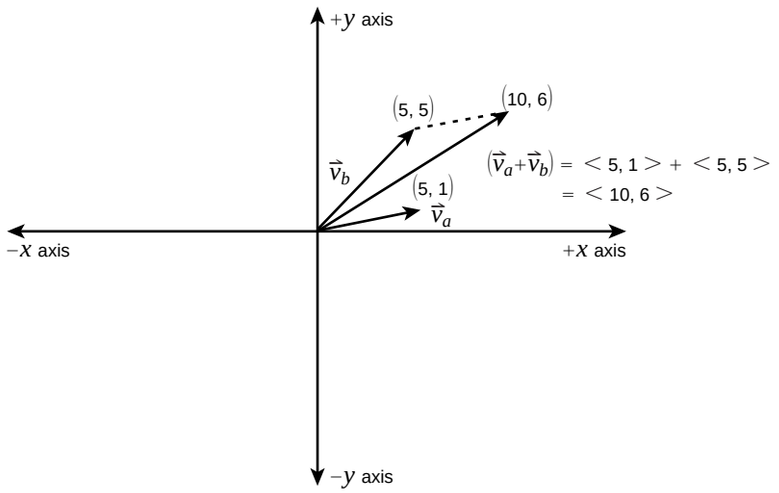


FIGURE 5.12 Vector addition setup.

Function Prototype(s):

```
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);

VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR3D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

Purpose:

`VECTOR*D VECTOR*D_Add()` is the stack version of the vector addition function that returns the result on the stack.

Example:

```
VECTOR2D v1 = {1,2};
VECTOR2D v2 = {5,6};

// add them and store result
VECTOR2D vsum = VECTOR2D_Add(&v1, &v2);
```

Function Prototype(s):

```
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vdiff);

void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vdiff);

void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vdiff);
```

Playrix Ex. 1023, page 434

08 0672318350 CH05 05•02•2003 2:37 PM Page 410

Purpose:

`void VECTOR*D_Sub()` subtracts (`vb-va`) as shown in Figure 5.13, and stores the result in `vdiff`.

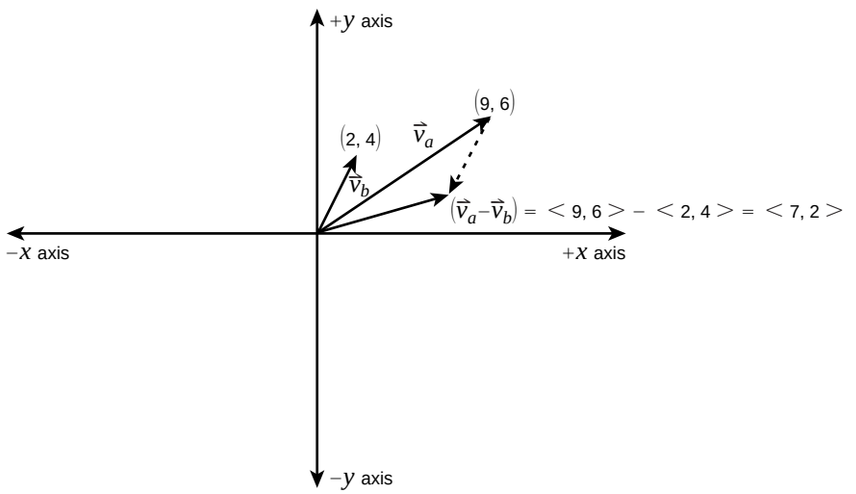


FIGURE 5.13 Vector subtraction setup.

Example:

```
VECTOR3D v1 = {1,2,3};
VECTOR3D v2 = {5,6,7};
VECTOR3D vdiff; // holds difference

// subtract em
VECTOR3D_Sub(&v1, &v2, &vdiff);
```

Function Prototype(s):

```
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);

VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR3D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

Purpose:

`VECTOR*D VECTOR*D_Sub()` is the stack version of the vector subtraction function that returns the result on the stack.

Playrix Ex. 1023, page 435

08 0672318350 CH05 05•02•2003 2:37 PM Page 411

Example:

```
VECTOR2D v1 = {1,2};
VECTOR2D v2 = {5,6};

// subtract them and store result
VECTOR2D vdiff = VECTOR2D_Sub(&v1, &v2);
```

Function Prototype(s):

```
void VECTOR2D_Scale(float k, VECTOR2D_PTR va, VECTOR2D_PTR vscaled);

void VECTOR3D_Scale(float k, VECTOR3D_PTR va, VECTOR3D_PTR vscaled);

void VECTOR4D_Scale(float k, VECTOR4D_PTR va, VECTOR4D_PTR vscaled);
```

Purpose:

`void VECTOR*D_Scale()` scales the sent vector `va` by `k` as shown in Figure 5.14, and stores the result (`k*va`) into `vscaled`.

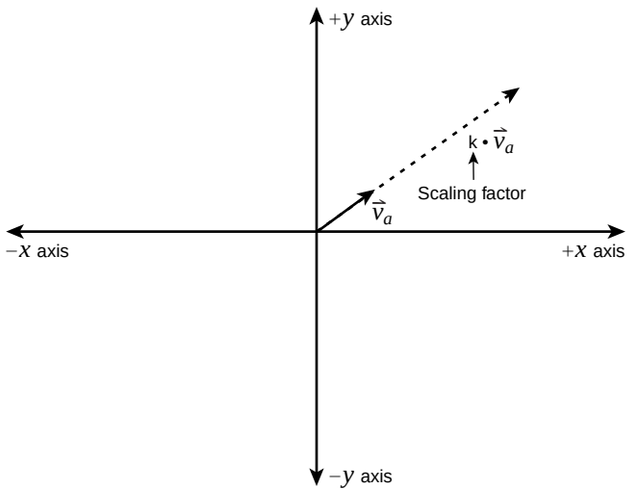


FIGURE 5.14 Vector scaling setup.

Example:

```
VECTOR3D v1 = {1,1,1};
VECTOR3D vs;
```

Playrix Ex. 1023, page 436

08 0672318350 CH05 05•02•2003 2:37 PM Page 412


```
// scale v1 by 50
VECTOR3D_Scale(50, &v1, &vs);

// invert v1
VECTOR3D_Scale(-1, &v1, &vs);
```

Function Prototype(s):

```
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);

float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);

float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

Purpose:

`float VECTOR*D_Dot()` computes the dot product (`va.vb`) and returns the scalar result. Note that for the `VECTOR4D` version, the w coordinate is disregarded.

Example:

```
VECTOR4D v1={1,0,0,1}; // the unit x-axis
VECTOR4D v2={0,1,0,1}; // the unit y-axis

// dot them together, the result better be 0!
float dp=VECTOR4D_Dot(&v1, &v2);
```

Function Prototype(s):

```
void VECTOR3D_Cross(VECTOR3D_PTR va,VECTOR3D_PTR vb,VECTOR3D_PTR vn);

void VECTOR4D_Cross(VECTOR4D_PTR va,VECTOR4D_PTR vb,VECTOR4D_PTR vn);
```

Purpose:

`void VECTOR*D_Cross()` computes the cross product (`va` x `vb`) as shown in Figure 5.15; that is, it computes the vector perpendicular to both `va` and `vb` and returns the result in `vn`. In the `VECTOR4D` case, w is disregarded as usual.

Example:

```
VECTOR3D vx = {1,0,0};
VECTOR3D vy = {0,1,0};
VECTOR3D vcross;

// cross the x-axis with the y-axis, the result should be
// the z-axis
VECTOR3D_Cross(&vx, &vy, &vcross);
```

Playrix Ex. 1023, page 437

08 0672318350 CH05 05•02•2003 2:37 PM Page 413

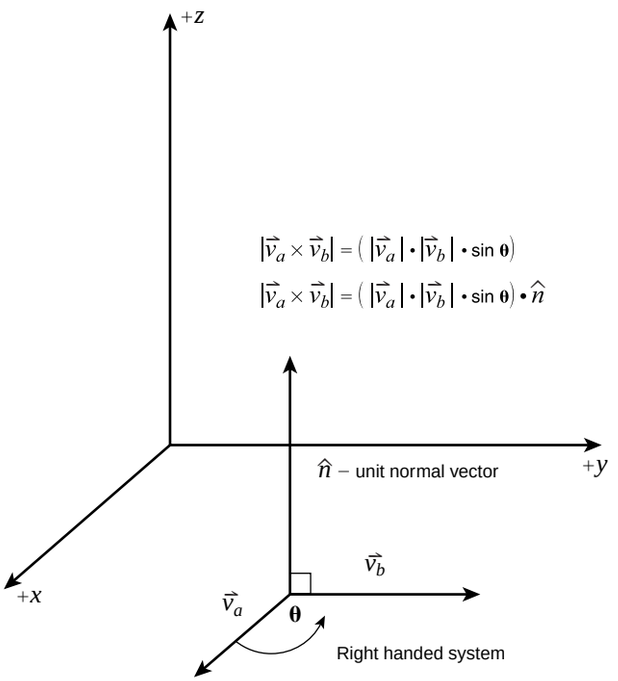


FIGURE 5.15 Vector cross product.

Function Prototype(s):

```
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

Purpose:

`VECTOR*D VECTOR*D_Cross()` computes the cross product (`va` x `vb`); that is, it computes the vector perpendicular to both `va` and `vb` and returns the result on the stack. In the `VECTOR4D` case, w is disregarded as usual.

Example:

```
VECTOR3D vx = {1,0,0};
VECTOR3D vy = {0,1,0};

// cross the x-axis with the y-axis, the result should be
// the z-axis
VECTOR3D vcross = VECTOR3D_Cross(&vx, &vy);
```

Playrix Ex. 1023, page 438

08 0672318350 CH05 05•02•2003 2:37 PM Page 414

Function Prototype(s):

```
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR4D_Length(VECTOR4D_PTR va);
```

Purpose:

`float VECTOR*D_Length()` computes the length of the sent vector using the standard square root of the sum of squares algorithm.

Example:

```
VECTOR2D v1={3,4};

// compute length, should be 5
float length = VECTOR3D_Length(&v1);
```

Function Prototype(s):

```
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);

float VECTOR3D_Length_Fast(VECTOR3D_PTR va);

float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
```

Purpose:

`float VECTOR*D_Length_Fast()` computes the length of the sent vector by using a Taylor series approximation. Accurate to within a few percent, but 10 times faster than `VECTOR*D_Length()` function.

Example:

```
VECTOR2D v1={3,4};

// compute length, should be 5
float length = VECTOR3D_Length_Fast(&v1);
```

Function Prototype(s):

```
void VECTOR2D_Normalize(VECTOR2D_PTR va);

void VECTOR3D_Normalize(VECTOR3D_PTR va);

void VECTOR4D_Normalize(VECTOR4D_PTR va);
```

Playrix Ex. 1023, page 439

08 0672318350 CH05 05•02•2003 2:37 PM Page 415

Purpose:

`void VECTOR*D_Normalize()` normalizes the vector `va`; that is, makes it a unit vector by dividing each component by the length of `va`. Note that `va` is modified.

Example:

```
VECTOR2D v={5,6};

// normalize v, v = v/¦v¦
VECTOR2D_Normalize(&v);
```

Function Prototype(s):

```
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);

void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);

void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
```

Purpose:

`void VECTOR2D_Normalize()` normalizes the vector `va` and stores it in `vn`.

Example:

```
VECTOR3D v={1,2,3};
VECTOR3D vn;

// vn = v/¦vn¦
VECTOR3D_Normalize(&v, &vn);
```

Function Prototype(s):

```
void VECTOR2D_Build(VECTOR2D_PTR init, VECTOR2D_PTR term, VECTOR2D_PTR result);

void VECTOR3D_Build(VECTOR3D_PTR init, VECTOR3D_PTR term, VECTOR3D_PTR result);

void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term, VECTOR4D_PTR result);
```

Purpose:

`void VECTOR*D_Build()` builds a vector from `init->term` and stores it in `result`. This is a good example of a function that works well using the data type alias `POINT*D` because in most cases you'll want to create a vector from two points.

Playrix Ex. 1023, page 440

08 0672318350 CH05 05•02•2003 2:37 PM Page 416

Example:

```
POINT3D p1={1,2,3}, p2={4,5,6};
VECTOR3D v12; // use to store the result

// create a vector from point p1 to p2
VECTOR3D_Build(&p1, &p2, &v);
```

Note that we don't need to cast the types because points and vectors of the same dimension have the same types and data footprints. But, if you want to create a vector from two other vectors, you can do so, of course.

Function Prototype(s):

```
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);

float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);

float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

Purpose:

`float VECTOR*D_CosTh()` computes the cosine of the angle between the two vectors `va` and `vb`, respectively, as shown in Figure 5.16.

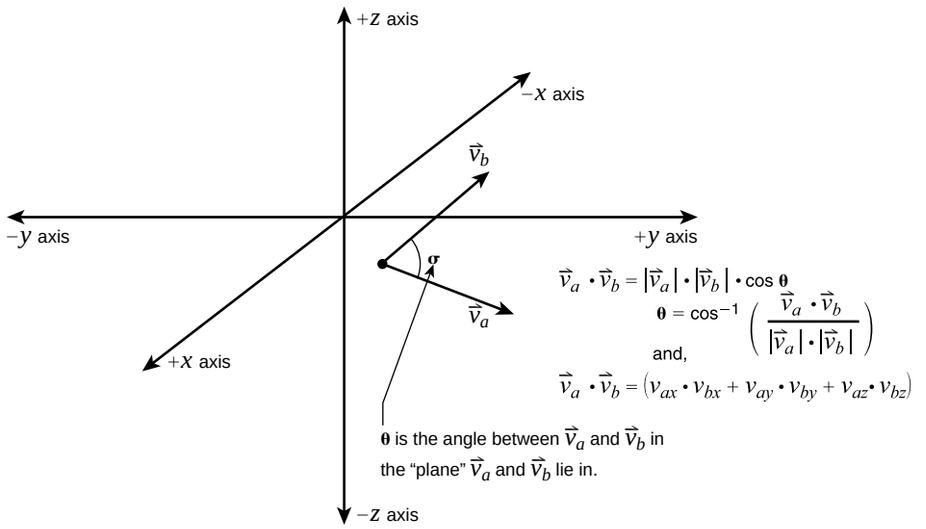


FIGURE 5.16 The dot product and interior vector angle.

Playrix Ex. 1023, page 441

08 0672318350 CH05 05•02•2003 2:37 PM Page 417

In many 3D algorithms, the product formula

$\mathbf{u}.\mathbf{v} = |\mathbf{u}|*|\mathbf{v}| * \cos(\theta)$

is used to solve for θ; that is, the angle between the two vectors, **u** and **v,** using the following simplification:

$\theta = \cos^{-1}[(\mathbf{u} . \mathbf{v})/(|\mathbf{u}|*|\mathbf{v}|)]$

However, many times we don't need to go that far and the cos(θ) term will suffice:

$\cos(\theta) = [(\mathbf{u} . \mathbf{v})/(|\mathbf{u}|*|\mathbf{v}|)]$

That's what this function is all about.

TRICK

There are many opportunities in game programming to exploit math and speed things up as in the preceding example. Another example is this: Say you have a number of objects, and when the objects get within 100 units of the black hole, you want them to get sucked in. You might blindly use the distance function and `sqrt()` function to compute the distance. If you're a little more clever though, you'll realize there's no need to use the exact distance—why not use the square of the distance? That is, use $d^2 = (x^2+y^2+z^2)$, and instead of 100 use (100*100) as the decision variable! This way, you lose the `sqrt()` function. The moral is that if $|x| < |y|$, $|x^2| < |y^2|$. But watch out for overflow!

Example:

```
VECTOR3D u = {0,1,0}; // the y-axis
VECTOR3D v = {1,1,0}; // a vector with a slope of 45 degrees

// find the angle between u and v
VECTOR3D_CosTh(&u, &v);
```

NOTE

This function is very useful in hidden surface removal where we need to detect if the angle between the view direction and the surface normal is acute or obtuse.

Function Prototype(s):

```
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);

void VECTOR3D_Print(VECTOR3D_PTR va, char *name);

void VECTOR4D_Print(VECTOR4D_PTR va, char *name);
```

Playrix Ex. 1023, page 442

08 0672318350 CH05 05•02•2003 2:37 PM Page 418

Purpose:

`void VECTOR*D_Print()` prints out a vector in "pretty form" along with a sent character name. This is used only as a debugging feature and writes the output to the error file previously opened with a call to `Open_Error_File()`.

Example:

```
// open the error file
Open_Error_File("error.txt");

VECTOR3D v={1,2,3};

// print the vector
VECTOR3D_Print(&v,"Velocity");

// close the error file
Close_Error_File();
```

File dump of ERROR.TXT:

```
Opening Error Output File (error.txt) on Sat Jan 29 22:36:00.520, 2000

Velocity=[1.000000, 2.000000, 3.000000, ]

Closing Error Output File.
```

Matrix Support Functions

The next set of functions handles matrix mathematics and transformations. In general, a 3D engine uses 3×3 or 4×4 matrices depending on what you want to do, but sometimes we cheat and use a 4×3 and make assumptions about the last column and other tricks like that. Remember, we have both 3D and 4D homogenous points, so even though at some point we'll optimize the matrix math a lot, it doesn't hurt to generalize at this early stage. Moreover, although I've tried to think of all the cases of matrix multiplication and transformations we might need, I'm sure I've left some out—but we can always add them later. Finally, there is support for 2×2, 3×3, and 4×4 in general, but some of the older 2x2 support was in the `T3DLIB1.CPP¦H` modules.

NOTE

Although the matrix library is very similar to the vector library inasmuch as there is support for various dimensional data types, all with similar function names, there are enough differences that I've decided to separate each functional set in the reference, because some functions on higher dimensions sometimes work differently. Hence, we'll cover 2×2, 3×3, and 4×4, in that order.

Playrix Ex. 1023, page 443

08 0672318350 CH05 05•02•2003 2:37 PM Page 419

Function Prototype:

```
void Mat_Init_2X2(MATRIX2X2_PTR ma, float m00, float m01, float m10, float m11);
```

Purpose:

`void Mat_Init_2X2()` initializes the matrix `ma` with the sent floating-point values in row major form.

Example:

```
MATRIX2X2 ma;

// initialize matrix to identity
Mat_Init_2X2(&ma, 1, 0, 0, 1);
```

Function Prototype:

```
void Mat_Add_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR msum);
```

Purpose:

`void Mat_Add_2X2()` adds the matrices (`ma+mb`) and stores the result in `msum`.

Example:

```
MATRIX2X2 m1 = {1,2, 3,4};
MATRIX2X2 m2 = {4,9, 5,6};
MATRIX2X2 msum;

// add em
Mat_Add_2X2(&m1, &m2, &msum);
```

Function Prototype:

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR mprod);
```

Function Listing Reference:

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR mprod)
{
// this function multiplies two 2x2 matrices together and
// and stores the result in mprod
mprod->M00 = ma->M00*mb->M00 + ma->M01*mb->M10;
mprod->M01 = ma->M00*mb->M01 + ma->M01*mb->M11;

mprod->M10 = ma->M10*mb->M00 + ma->M11*mb->M10;
mprod->M11 = ma->M10*mb->M01 + ma->M11*mb->M11;

} // end Mat_Mul_2X2
```

Playrix Ex. 1023, page 444

08 0672318350 CH05 05•02•2003 2:37 PM Page 420

Purpose:

`void Mat_Mul_2X2()` matrix multiplies (`ma*mb`) and stores the result in `mprod`. Also refer to the earlier code listing and note that the matrices are explicitly multiplied out rather than using loop operations. This is optimal.

Example:

```
MATRIX2X2 m1 = {1,2, 3,4};
MATRIX2X2 m2 = {1,0, 0,1};
MATRIX2X2 mprod;


// multiply them, note that m1*m2 = m1 since m2=I
Mat_Mul_2X2(&m1, &m2, &mprod);
```

Function Prototype (from `T3DLIB1.CPP¦H`):

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod);
```

Function Listing Reference:

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod)
{
// this function multiplies a 1x2 matrix against a
// 3x2 matrix - ma*mb and stores the result
// using a dummy element for the 3rd element of the 1x2
// to make the matrix multiply valid i.e. 1x3 X 3x2

  for (int col=0; col<2; col++)
    {
    // compute dot product from row of ma
    // and column of mb

    float sum = 0; // used to hold result

    for (int index=0; index<2; index++)
       {
       // add in next product pair
       sum+=(ma->M[index]*mb->M[index][col]);
       } // end for index
```

Playrix Ex. 1023, page 445

08 0672318350 CH05 05•02•2003 2:37 PM Page 421


```
    // add in last element * 1
    sum+= mb->M[index][col];

    // insert resulting col element
    mprod->M[col] = sum;

    } // end for col

return(1);

} // end Mat_Mul_1X2_3X2
```

Purpose:

`int Mat_Mul_1X2_3X2()` is a specialized function used to multiply a 1×2 matrix (basically a 2D point) against a 3×3 matrix that represents a rotation and a translation. The operation is mathematically undefined because the inner dimensions don't match. However, if we assume that the 1×2 is really 1×3 with the last element equal to 1, the multiplication can be carried out. Figure 5.17 illustrates what's being multiplied. Additionally, take a look at the code listing. This function is useful when transforming 2D points, but you want to save space on their representation and still want to support translation and homogenous coordinates.

Example:

```
MATRIX1X2 p1={5,5}, p2; // the point

// transformation rotation plus translation
MATRIX3X2 m = {cos(th), sin(th),
       -sin(th), cos(th),
        dx,   dy};

// perform matrix multiply
Mat_Mul_1X2_3X2(&p1, &m, &p2);
```

TIP

The function itself is rather slow. We could speed it up by unrolling the loops, but it's much clearer this way and you can see its iterative nature. Remember, never optimize until you need to, and start with your algorithms before you start looking at the actual code itself. Then, once you have optimal algorithms and code, if you really need the last step, inline assembly language is in order.

Playrix Ex. 1023, page 446

08 0672318350 CH05 05•02•2003 2:37 PM Page 422

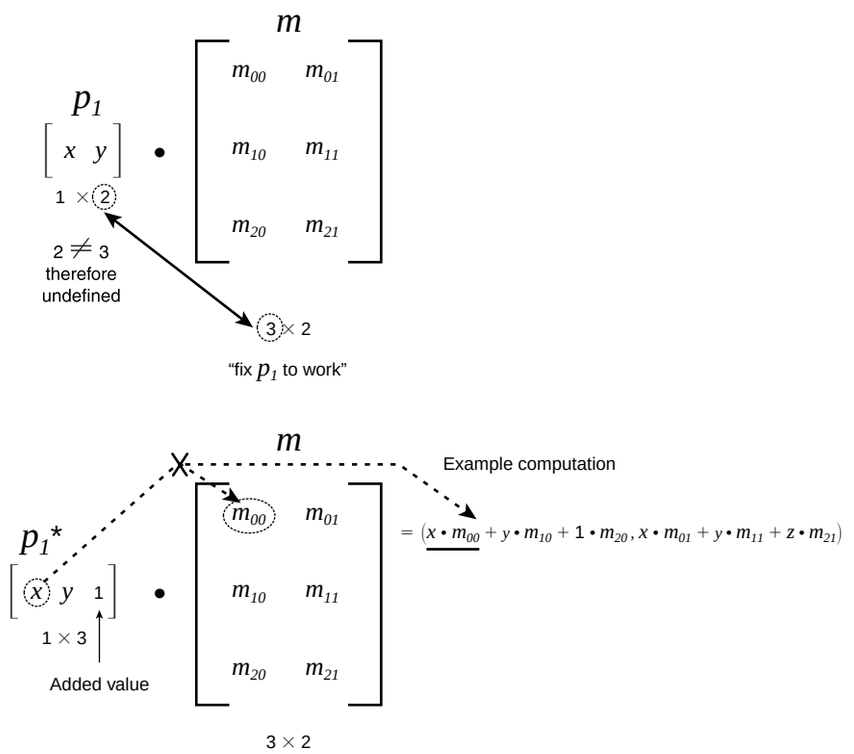


FIGURE 5.17 The methodology of undefined vector-matrix multiplication for the (1×2)*(3×2) case.

Function Prototype:

```
float Mat_Det_2X2(MATRIX2X2_PTR m);
```

Purpose:

`float Mat_Det_2X2()` computes the determinate of the matrix `m` and returns it on the stack.

Example:

```
MATRIX2x2 m = {1,2,4,8}; // this matrix is singular

// compute determinate
float det = Mat_Det_2X2(&m);
```

Function Prototype:

```
int Mat_Inverse_2X2(MATRIX2X2_PTR m, MATRIX2X2_PTR mi);
```

Playrix Ex. 1023, page 447

08 0672318350 CH05 05•02•2003 2:37 PM Page 423

Purpose:

`int Mat_Inverse_2X2()` computes the inverse of `m` if it exists and returns it in `mi`. If the inverse exists, the function returns a `1`; otherwise, it returns a `0` and `mi` is undefined.

Example:

```
MATRIX2X2 mA = {1,2, 4, -3};
MATRIX2X2 mI;

// compute the inverse of A
if (Mat_Inverse_2X2(&mA, &mI))
  {
  // inverse exists...
  } // end if
else
  {
  // inverse doesn't exist
  } // end else
```


Function Prototype:

```
void Print_Mat_2X2(MATRIX2X2_PTR ma, char *name);
```

Purpose:

`void Print_Mat_2X2()` prints out the matrix in "pretty" format along with the sent string in `name` as an identifier. All output, of course, goes to the error file previously opened with `Open_Error_File()`.

Example:

```
MATRIX2X2 m = {1,2,3,4};

// open the error file and write to the screen
Open_Error_File("", stdout);

Print_Mat_2X2(&m,"Matrix m");

// close the error file
Close_Error_File();
```

Function Prototype:

```
void Mat_Init_3X3(MATRIX3X3_PTR ma,
         float m00, float m01, float m02,
         float m10, float m11, float m12,
         float m20, float m21, float m22);
```

Playrix Ex. 1023, page 448

08 0672318350 CH05 05•02•2003 2:37 PM Page 424

Purpose:

`void Mat_Init_3X3()` initializes the matrix `ma` with the sent floating-point values in row major form.

Example:

```
MATRIX3X3 ma;

// initialize matrix to identity
Mat_Init_3X3(&ma, 1,0,0, 0,1,0, 0,0,1);
```

Function Prototype (from `T3DLIB1.CPP¦H`):

```
inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
            float m00, float m01,
            float m10, float m11,
            float m20, float m21);
```

Purpose:

`void Mat_Init_3X2()` initializes the matrix `ma` with the sent floating-point values in row major form.

Example:

```
MATRIX3X2 ma;

// initialize minor upper left hand 2x2
// matrix to identity matrix
Mat_Init_3X2(&ma, 1,0, 0,1, 0,0);
```

Function Prototype:

```
void Mat_Add_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb, MATRIX3X3_PTR msum);
```

Purpose:

`void Mat_Add_3X3()` adds the matrices (`ma+mb`) and stores the result in `msum`.

Example:

```
MATRIX3X3 m1 = {1,2,3,4,5,6,7,8,9};
MATRIX3X3 m2 = {4,9,7, -1,5,6, 2,3,4};
MATRIX3X3 msum;

// add em
Mat_Add_3X3(&m1, &m2, &msum);
```

Playrix Ex. 1023, page 449

08 0672318350 CH05 05•02•2003 2:37 PM Page 425

Function Prototype:

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb,VECTOR3D_PTR vprod);
```

Purpose:

`void Mat_Mul_VECTOR3D_3X3()` multiplies the 1×3 row vector `va` against the 3×3 matrix `mb`, and stores the result in the 1x3 row vector `vprod`. This is shown in Figure 5.18. Basically, this function performs a point or vector matrix multiplication.

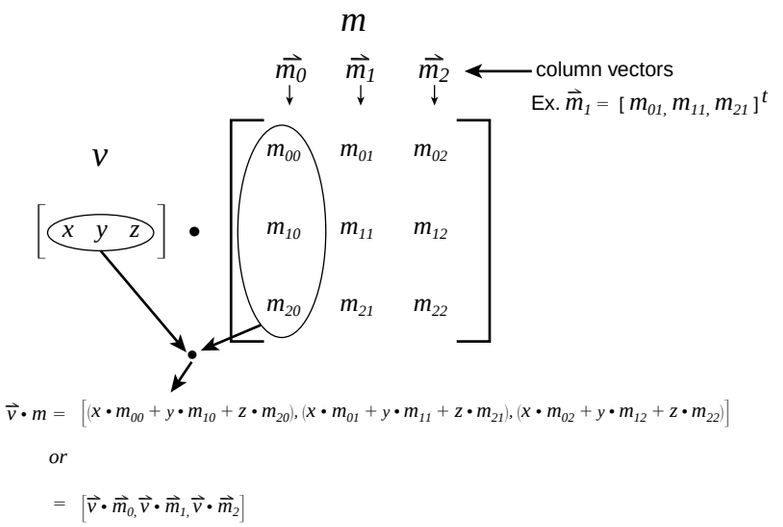


FIGURE 5.18 3D vector multiplied with 3×3 matrix.

Example:

```
VECTOR3D v={x,y,1}, vt;
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};

// multiply v*m
Mat_Mul_VECTOR3D_3X3(&v, &m, &vt);
```

Can you figure out what this transform does? If we pretend that the `VECTOR3D` is really a 2D homogenous point, the transformation translates the point to

```
vt.x = v.x+xt;
vt.y = v.y+yt;
vt.z = 1;
```

Neat, huh? But a new Dodge Viper would be neater <VBG>.

Playrix Ex. 1023, page 450

08 0672318350 CH05 05•02•2003 2:37 PM Page 426

Function Prototype (from T3DLIB1.CPP¦H):

```
int (MATRIX1X3_PTR ma,
          MATRIX3X3_PTR mb,
          MATRIX1X3_PTR mprod);
```

Purpose:

`int Mat_Mul_1X3_3X3()` multiplies a 1×3 matrix—basically a row vector—against a 3×3 matrix. This function is almost identical to `Mat_Mul_VECTOR3D_3X3()`. Only the types are different.

Example:

```
MATRIX1X3 v={x,y,1}, vt;
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};

// multiply v*m
Mat_Mul_1X3_3X3(&v, &m, &vt);
```

Function Prototype (from T3DLIB1.CPP¦H):

```
int Mat_Mul_3X3(MATRIX3X3_PTR ma,
        MATRIX3X3_PTR mb,
        MATRIX3X3_PTR mprod);
```

Purpose:

`void Mat_Mul_3X3()` matrix multiplies (`ma*mb`) and stores the result in `mprod`.

Example:

```
MATRIX3X3 m1 = {1,2,3, 4,5,6, 7,8,9};
MATRIX3X3 m2 = {1,0,0, 0,1,0, 0,0,1};
MATRIX3X3 mprod;

// multiply them, note that m1*m2 = m1 since m2=I
Mat_Mul_3X3(&m1, &m2, &mprod);
```

Function Prototype:

```
float Mat_Det_3X3(MATRIX3X3_PTR m);
```

Purpose:

`float Mat_Det_3X3()` computes the determinate of the matrix `m` and returns it on the stack.

Playrix Ex. 1023, page 451

08 0672318350 CH05 05•02•2003 2:37 PM Page 427

Example:

```
MATRIX3x3 m = {1,2,0, 4,8,9, 2,5,7};

// compute determinate
float det = Mat_Det_3X3(&m);
```

Function Prototype:

```
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
```

Purpose:

`int Mat_Inverse_3X3()` computes the inverse of `m` if it exists and returns it in `mi`. If the inverse exists, the function returns a `1`; otherwise, it returns a `0` and `mi` is undefined.

Example:

```
MATRIX3X3 mA = {1,2,9, 4,-3,6, 1,0,5};
MATRIX3X3 mI;

// compute the inverse of A
if (Mat_Inverse_3X3(&mA, &mI))
  {
  // inverse exists…
  } // end if
else
  {
  // inverse doesn't exist
  } // end else
```

Function Prototype:

```
void Print_Mat_3X3(MATRIX3X3_PTR ma, char *name);
```

Purpose:

`void Print_Mat_3X3()` prints out the matrix in "pretty" format along with the sent string in `name` as an identifier. All output, of course, goes to the error file previously opened with `Open_Error_File()`.

Example:

```
MATRIX3X3 m = {1,2,3, 4,5,6, 7,8,9};

// open the error file and write to a file
Open_Error_File("error.txt");
```

Playrix Ex. 1023, page 452

08 0672318350 CH05 05•02•2003 2:37 PM Page 428


```
Print_Mat_3X3(&m,"Matrix m");

// close the error file
Close_Error_File();
```

Function Prototype:

```
void Mat_Init_4X4(MATRIX4X4_PTR ma,
      float m00, float m01, float m02, float m03,
      float m10, float m11, float m12, float m13,
      float m20, float m21, float m22, float m23,
      float m30, float m31, float m32, float m33);
```

Purpose:

`void Mat_Init_4X4()` initializes the matrix `ma` with the sent floating-point values in row major form.

Example:

```
MATRIX4X4 ma;

// initialize matrix to identity
Mat_Init_4X4(&ma, 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1);
```

Function Prototype:

```
void Mat_Add_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR msum);
```

Purpose:

`void Mat_Add_4X4()` adds the matrices (`ma+mb`) and stores the result in `msum`.

Example:

```
MATRIX4X4 m1 = {1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16};
MATRIX4X4 m2 = {4,9,7,3, -1,5,6,7, 2,3,4,5, 2,0,5,3};
MATRIX4X4 msum;

// add em
Mat_Add_4X4(&m1, &m2, &msum);
```

Function Prototype:

```
void Mat_Mul_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR mprod);
```

Purpose:

`void Mat_Mul_4X4()` matrix multiplies (`ma*mb`) and stores the result in `mprod`.

Playrix Ex. 1023, page 453

08 0672318350 CH05 05•02•2003 2:37 PM Page 429

Example:

```
MATRIX4X4 m1 = {1,2,3,4, 4,5,6,7, 7,8,9,10, 11,12,13,14};
MATRIX4X4 m2 = {1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1};
MATRIX4X4 mprod;

// multiply them, note that m1*m2 = m1 since m2=I
Mat_Mul_4X4(&m1, &m2, &mprod);
```

Function Prototype:

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma, MATRIX4X4_PTR mb, MATRIX1X4_PTR mprod);
```

Function Listing Reference:

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma,
           MATRIX4X4_PTR mb,
           MATRIX1X4_PTR mprod)
{
// this function multiplies a 1x4 matrix against a
// 4x4 matrix - ma*mb and stores the result
// no tricks or assumptions here, just a straight multiply

  for (int col=0; col<4; col++)
    {
    // compute dot product from row of ma
    // and column of mb
    float sum = 0; // used to hold result

    for (int row=0; row<4; row++)
       {
       // add in next product pair
       sum+=(ma->M[row] * mb->M[row][col]);
       } // end for index

    // insert resulting col element
    mprod->M[col] = sum;

    } // end for col

} // end Mat_Mul_1X4_4X4
```


Playrix Ex. 1023, page 454

08 0672318350 CH05 05•02•2003 2:37 PM Page 430

Purpose:

`void Mat_Mul_1X4_4X4()` multiplies a 1×4 row matrix against a 4×4 matrix and computes the product `mprod = (ma*mb)`. The function makes no assumptions about homogenous coordinates and so forth, so it's a straight matrix multiply as shown in the code listing. Also, note that we can speed this function up by explicitly performing all the math rather than using loops.

Example:

```
MATRIX1X4 v={x,y,z,1}, vt;
MATRIX4X4 m = {1,0,0,0 0,1,0,0, 0,0,1,0, xt,yt,zt,1};

// multiply v*m, this translates the 3D point x,y,z
Mat_Mul_1X4_4X4(&v, &m, &vt);
```

Function Prototype:

```
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va, MATRIX4X4_PTR mb, VECTOR3D_PTR vprod);
```

Purpose:

`void Mat_Mul_VECTOR3D_4X4()` multiplies a 3D vector against a 4×4 matrix. However, to perform the multiplication, the function assumes that there's a phantom fourth element in `va` that's equal to 1.0. Moreover, when the multiplication completes, the result is a 3D vector, rather than a 4D vector. The bottom line is that this function is used to transform 3D points with 4×4 matrices that contain translation values and assume homogenous coordinates. Because a 3D point with only three elements [x y z] doesn't have enough space for the homogenous component w, we assume it's there and set it to 1.0. That was a mouthful! Figure 5.19 might clear this up for you a bit.

Example:

```
VECTOR3D v={10,10,10}, vrt;

// rotate around the x-axis and translate (tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// do the transform
Mat_Mul_VECTOR3D_4X4(&v, &m, &vrt);
```

Playrix Ex. 1023, page 455

08 0672318350 CH05 05•02•2003 2:37 PM Page 431

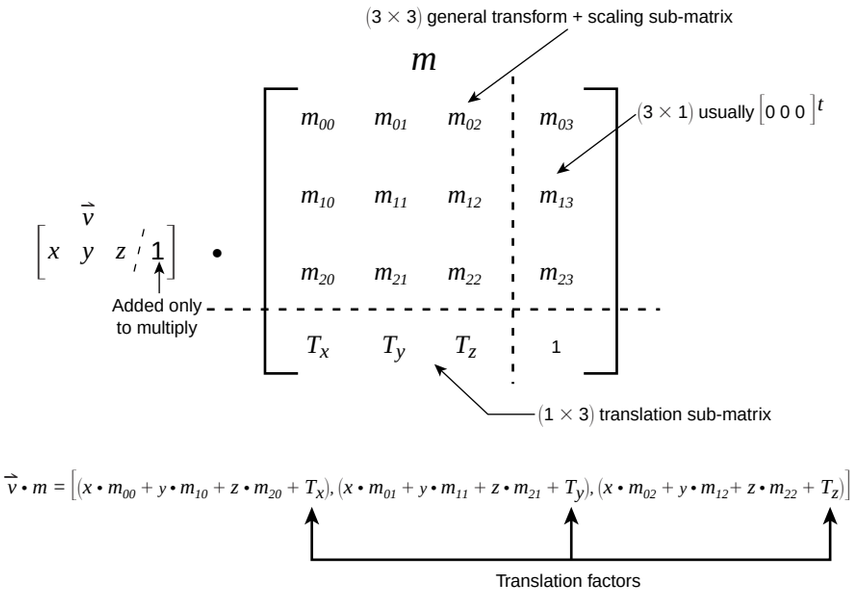


FIGURE 5.19 3D vector multiplied against a 4×4 matrix.

Also, we could've defined the vectors as points made the same call because `VECTOR3D` and `POINT3D` are aliases:

```
POINT3D p1={10,10,10}, p2;

// rotate around the x-axis and translate (tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// do the transform
Mat_Mul_VECTOR3D_4X4(&p1, &m, &p2);
```

Function Prototype:

```
void Mat_Mul_VECTOR3D_4X3(VECTOR3D_PTR va, MATRIX4X3_PTR mb, VECTOR3D_PTR vprod);
```

Purpose:

`void Mat_Mul_VECTOR3D_4X3()` is very similar to `Mat_Mul_VECTOR3D_4X4()`, but instead it multiplies a 3D vector against a 4×3 matrix rather than a 4×4 matrix. However, it also assumes that there's a phantom fourth element in `va` that's equal to 1.0 to carry out the

Playrix Ex. 1023, page 456

08 0672318350 CH05 05•02•2003 2:37 PM Page 432

multiplication. And because there are only three columns in the 4×3 matrix, the multiplication stops naturally rather than being cut off artificially as in `Mat_Mul_VECTOR3D_4X4()`.

Example:

```
POINT3D p1={10,10,10}, p2;

// rotate around the x-axis and translate (tx,ty,tz)
// note that we don't have to muck with the last column
// which is always [0 0 0 1]t anyway
MATRIX4X3 m = {1, 0, 0,
        0, cos(th), sin(th),
        0,-sin(th), cos(th),
        tx, ty, tz};

// do the transform
Mat_Mul_VECTOR3D_4X3(&p1, &m, &p2);
```

Function Prototype:

```
void Mat_Mul_VECTOR4D_4X4(VECTOR4D_PTR va, MATRIX4X4_PTR mb, VECTOR4D_PTR vprod);
```

Purpose:

`void Mat_Mul_VECTOR4D_4X4()` multiplies the 1×4 row vector `va` against the 4×4 matrix `mb`, and stores the result in the 1×4 row vector `vprod`. The function assumes nothing special and simply does a 1×4 * 4×4 matrix multiplication.

Example:

```
VECTOR4D v={10,10,10,1}, vrt;

// rotate around the x-axis and translate (tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// do the transform
Mat_Mul_VECTOR4D_4X4(&v, &m, &vrt);
```

Note that the result is **vrt** = <x', y', z', 1>. That is, the w, or homogenous component, will be equal to 1.0.

Function Prototype:

```
void Mat_Mul_VECTOR4D_4X3(VECTOR4D_PTR va, MATRIX4X4_PTR mb, VECTOR4D_PTR vprod);
```

Playrix Ex. 1023, page 457

08 0672318350 CH05 05•02•2003 2:37 PM Page 433

Purpose:

`void Mat_Mul_VECTOR4D_4X3()` is very similar to `Mat_Mul_VECTOR3D_4X3()`, but instead it multiplies a 4D vector against a 4×3 matrix rather than a 3D vector. And because there are only three columns in the 4×3 matrix, the multiplication stops naturally rather than being cut off artificially, as in `Mat_Mul_VECTOR3D_4X4()`. Finally, the w component from `va` is copied to `vprod`. Or, in other words, the last column of the 4×3 is assumed to be [0 0 0 1]^t.

Example:

```
POINT4D p1={10,10,10,1}, p2;

// rotate around the x-axis and translate (tx,ty,tz)
// note that we don't have to muck with the last column
// which is always [0 0 0 1]t anyway
MATRIX4X3 m = {1, 0, 0,
        0, cos(th), sin(th),
        0,-sin(th), cos(th),
        tx, ty, tz};

// do the transform
Mat_Mul_VECTOR4D_4X3(&p1, &m, &p2);
```

Function Prototype:

```
int Mat_Inverse_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mi);
```

Purpose:

`int Mat_Inverse_4X4()` computes the inverse of `m` if it exists and returns it in `mi`. If the inverse exists, the function returns a `1`; otherwise, it returns a `0` and `mi` is undefined. Also, this function works only on matrices where the last column is [0 0 0 1]^t.

Example:

```
// note the last column is [0 0 0 1]t
MATRIX4X4 mA = {1, 2,9,0,
        4,-3,6,0,
        1, 0,5,0,
        2, 3,4,1};

MATRIX4X4 mI;

// compute the inverse of A
if (Mat_Inverse_4X4(&mA, &mI))
  {
  // inverse exists…
```

Playrix Ex. 1023, page 458

08 0672318350 CH05 05•02•2003 2:37 PM Page 434


```
  } // end if
else
  {
  // inverse doesn't exist
  } // end else
```

Function Prototype:

```
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);
```

Purpose:

`void Print_Mat_4X4()` prints the matrix out in "pretty" format along with the sent string in `name` as an identifier. All output, of course, goes to the error file previously opened with `Open_Error_File()`.

Example:

```
MATRIX4X4 m = {1,2,3,4,
        5,6,7,8,
        9,10,11,12,
        13,14,15,16};

// open the error file and write to a file
Open_Error_File("error.txt");

Print_Mat_4X4(&m,"Matrix m");

// close the error file
Close_Error_File();
```

2D and 3D Parametric Line Support Functions

When I started writing the math engine, I didn't think at all that I would add parametric line support into the engine. Why? Because most of the time, you would just manually code parametric lines in the algorithms where you need them with a vector and a parameter `t`. However, 9 times out of 10, you would use a vector and a parameter `t`, so why not encapsulate that in a structure (or class without methods) and then build on that for common functions such as intersection, and so forth? Hence, the math engine has support for both 2D and 3D parametric lines, along with a fairly complete set of functions to work with them.

To refresh your memory, here's the structure for a 2D parametric line:

```
typedef struct PARMLINE2D_TYP
{
POINT2D p0; // start point of parametric line
```

Playrix Ex. 1023, page 459

08 0672318350 CH05 05•02•2003 2:37 PM Page 435


```
POINT2D p1; // end point of parametric line
VECTOR2D v; // direction vector of line segment
       // ¦v¦=¦p0->p1¦
} PARMLINE2D, *PARMLINE2D_PTR;
```

The relationship of the values is shown in Figure 5.2. Note that the vector extends from **p0**->**p1**; that is, it's not normalized. And moving on, here's the structure for a 3D parametric line:

```
typedef struct PARMLINE3D_TYP
{
POINT3D p0; // start point of parametric line
POINT3D p1; // end point of parametric line
VECTOR3D v; // direction vector of line segment
       // ¦v¦=¦p0->p1¦
} PARMLINE3D, *PARMLINE3D_PTR;
```

The only difference between the 2D version and the 3D version is the z-axis; other than that, each structure holds the same info in the same form. So, let's take a look at the functions now.

Function Prototype:

```
void Init_Parm_Line2D(POINT2D_PTR p_init, POINT2D_PTR p_term, PARMLINE2D_PTR p);
```

Purpose:

`void Init_Parm_Line2D()` initializes a 2D parametric line structure based on the sent points and computes the vector between them. Note that the vector is generated internally to the function.

Example:

```
POINT2D p1 = {1,2}, p2 = {10,20};
PARMLINE2D p;

// create a parametric line from p1 to p2
Init_Parm_Line2D(&p1, &p2, &p);
```

Function Prototype:

```
void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t, POINT2D_PTR pt);
```

Playrix Ex. 1023, page 460

08 0672318350 CH05 05•02•2003 2:37 PM Page 436

Purpose:

`void Compute_Parm_Line2D()` evaluates a 2D parametric line at the value `t` and returns the value in the sent point. Note that when parameter `t = 0`, the parametric line is equal to the initial point `p1`; when parameter `t = 1`, the parametric line equals the terminal point `p2`. That is, the line is traced from **p1** to **p2** as 0 <= t <= 1.

Example:

```
POINT2D p1 = {1,2}, p2 = {10,20}, pt;
PARMLINE2D p;

// create a parametric line from p1 to p2
Init_Parm_Line2D(&p1, &p2, &p);

// now evaluate the parametric line when t=.5
// and store the resulting point in pt
Compute_Parm_Line2D(&p, 0.5, &pt);
```

Function Prototype:

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
              float *t1, float *t2);
```

Functional Listing:

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
              float *t1, float *t2)
{
// this function computes the intersection of the two parametric
// line segments the function returns true if the segments
// intersect and sets the values of t1 and t2 to the t values that
// the intersection occurs on the lines p1 and p2 respectively,
// however, the function may send back t value not in the range [0,1]
// this means that the segments don't intersect, but the lines that
// they represent do, thus a return of 0 means no intersection, a
// 1 means intersection on the segments and a 2 means the lines
// intersect, but not necessarily the segments and 3 means that
// the lines are the same, thus they intersect everywhere

// basically we have a system of 2d linear equations, we need
// to solve for t1, t2 when x,y are both equal (if that's possible)

// step 1: test for parallel lines, if the direction vectors are
// scalar multiples then the lines are parallel and can't possible
// intersect unless the lines overlap
```

Playrix Ex. 1023, page 461

08 0672318350 CH05 05•02•2003 2:37 PM Page 437


```
float det_p1p2 = (p1->v.x*p2->v.y - p1->v.y*p2->v.x);
if (fabs(det_p1p2) <= EPSILON_E5)
  {
  // at this point, the lines either don't intersect at all
  // or they are the same lines, in which case they may intersect
  // at one or many points along the segments, at this point we
  // will assume that the lines don't intersect at all, but later
  // we may need to rewrite this function and take into
  // consideration the overlap and singular point exceptions
  return(PARM_LINE_NO_INTERSECT);

  } // end if

// step 2: now compute the t1, t2 values for intersection
// we have two lines of the form
// p  = p0  + v*t, specifically
// p1  = p10  + v1*t1

// p1.x = p10.x + v1.x*t1
// p1.y = p10.y + v1.y*t1

// p2 = p20 + v2*t2

// p2.x = p20.x + v2.x*t2
// p2.y = p20.y + v2.y*t2
// solve the system when x1 = x2 and y1 = y2
// explained in chapter 4
*t1 = (p2->v.x*(p1->p0.y - p2->p0.y) - p2->v.y*(p1->p0.x - p2->p0.x))
   /det_p1p2;

*t2 = (p1->v.x*(p1->p0.y - p2->p0.y) - p1->v.y*(p1->p0.x - p2->p0.x))
   /det_p1p2;

// test if the lines intersect on the segments
if ((*t1>=0) && (*t1<=1) && (*t2>=0) && (*t2<=1))
  return(PARM_LINE_INTERSECT_IN_SEGMENT);
else
  return(PARM_LINE_INTERSECT_OUT_SEGMENT);

} // end Intersect_Parm_Lines2D
```


Playrix Ex. 1023, page 462

08 0672318350 CH05 05•02•2003 2:37 PM Page 438

Purpose:

`int Intersect_Parm_Lines2D()` computes the intersection of two parametric lines, `p1` and `p2`, and sends back the parametric values `t1` and `t2` where the intersection takes place on each line segment, respectively. The function has the following return values based on whether there's an intersection on the parametric segment on the lines projected from the segments.

```
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
```

Note that the function doesn't test whether the lines are co-linear because it would slow it down too much and there are too many cases (partial overlap, inclusion, point co-linearity). Therefore, it's up to you to check whether you need to know that. Or, you can add the functionality to the function. The thing to remember is that these parametric line segments are really finite segments, and all lines that aren't parallel intersect at some point, so the function determines the intersection regardless. However, if the `t` parameters coming back aren't in the range (0 <= `t` <= 1), you know that the intersection didn't occur on the original segment **p1->p2** of each line, respectively. Check out `DEMOII5_1.CPP¦EXE` on the CD; it allows you to define parametric lines and test whether they intersect.

Example:

```
// these lines intersect
POINT2D p1 = {1,1}, p2 = {9,8};
POINT2D p3 = {2,8}, p4 = {7,1};

PARMLINE2D pl1, pl2;

// create a parametric line from p1 to p2
Init_Parm_Line2D(&p1, &p2, &pl1);

// create a parametric line from p3 to p4
Init_Parm_Line2D(&p3, &p4, &pl2);

float t1=0, t2=0; // storage for parameters

// compute intersection
int intersection_type = Intersect_Parm_Lines2D(&pl1, &pl2, &t1, &t2);
```

Function Prototype:

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2, POINT2D_PTR pt);
```

Playrix Ex. 1023, page 463

08 0672318350 CH05 05•02•2003 2:37 PM Page 439

Purpose:

`int Intersect_Parm_Lines2D()` computes the intersection of the parametric lines, but rather then sending back the parameters `t1` and `t2`, the function returns the actual point of intersection. Of course, care needs to be taken and the return value reviewed to see the type of intersection just as in the previous version.

Example:

```
// these lines intersect
POINT2D p1 = {1,1}, p2 = {9,8};
POINT2D p3 = {2,8}, p4 = {7,1};

POINT2D pt; // storage for intersection

PARMLINE2D pl1, pl2;

// create a parametric line from p1 to p2
Init_Parm_Line2D(&p1, &p2, &pl1);

// create a parametric line from p3 to p4
Init_Parm_Line2D(&p3, &p4, &pl2);

// compute intersection
int intersection_type = int Intersect_Parm_Lines2D(&pl1, &pl2, &pt);
```


C++

Pure C programmers might be a bit bewildered by two functions with the same name. This is a C++ feature. Basically, the parameter footprint of each function is different; thus, as far as the compiler is concerned, the functions are different. This is called *function overloading*, and is accomplished with name mangling based on the parameter list to come up with two unique names. This is covered in Appendix D, "C++ Primer."

And now here are the 3D parametric line functions. Note there are fewer because I classed some of them along with the 3D plane functions.

Function Prototype:

```
void Init_Parm_Line3D(POINT3D_PTR p_init, POINT3D_PTR p_term, PARMLINE3D_PTR p);
```

Purpose:

`void Init_Parm_Line3D()` initializes a 3D parametric line structure based on the sent points and the vector between them. Note that the vector is generated internally to the function.

Playrix Ex. 1023, page 464

08 0672318350 CH05 05•02•2003 2:37 PM Page 440

Example:

```
POINT3D p1 = {1,2,3}, p2 = {10,20,30};
PARMLINE3D p;

// create a parametric line from p1 to p2
Init_Parm_Line3D(&p1, &p2, &p);
```

Function Prototype:

```
void Compute_Parm_Line3D(PARMLINE3D_PTR p, float t, POINT3D_PTR pt);
```

Purpose:

`void Compute_Parm_Line3D()` evaluates a 3D parametric line at the value `t` and returns the value in the sent point. Note that when parameter `t` = 0, the parametric line is equal to **p1**; when parameter `t` = 1, the parametric line equals **p2**. That is, the line is traced from **p1** to **p2** as 0 <= t <= 1.

Example:

```
POINT3D p1 = {1,2,3}, p2 = {10,20,30}, pt;
PARMLINE3D p;

// create a parametric line from p1 to p2
Init_Parm_Line3D(&p1, &p2, &p);

// now evaluate the parametric line when t=.5
// and store the resulting point in pt
Compute_Parm_Line3D(&p, 0.5, &pt);
```

3D Plane Support Functions

Although this might be a little premature, I decided to add some support for the abstract entity of 3D planes to the math engine. In reality, 99% of the time, we'll deal with closed polygons (that lie in planes), so these functions might need to be retrofitted later. Nevertheless, it's nice to be able to define planes and then work with them at least a little bit at this early stage.

I decided to go with the point-normal representation of a plane, which is shown in Figure 5.20.

```
nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0
```

where **n** = <nx,ny,nz>, and **p0** = <x0,y0,z0>.

Playrix Ex. 1023, page 465

08 0672318350 CH05 05•02•2003 2:37 PM Page 441

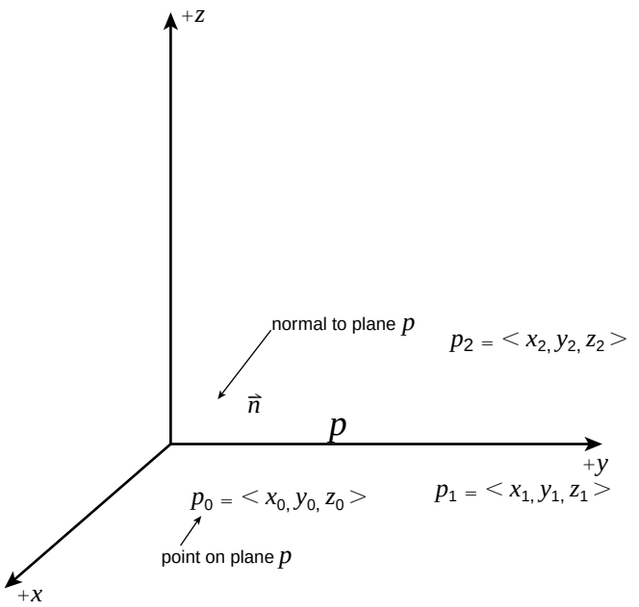


FIGURE 5.20 The point-normal-defined 3D plane.

Basically, in this format, we store a plane by means of its normal and a single point that lies on the plane. The normal need not be a unit normal. Once again, here's the structure for a 3D plane:

```
typedef struct PLANE3D_TYP
{
POINT3D p0; // point on the plane
VECTOR3D n; // normal to the plane (not necessarily a unit vector)
} PLANE3D, *PLANE3D_PTR;
```

Not much to it! Again, many of the functions in a math library just simplify repetitive calculations that you find yourself doing all the time, or ease the representation of mathematical or geometrical objects.

Function Prototype:

```
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
            VECTOR3D_PTR normal, int normalize);
```

Purpose:

`void PLANE3D_Init()` initializes a 3D plane with the sent point and normal. Additionally, the function can normalize the sent normal vector and make its length 1.0 for you. To do

Playrix Ex. 1023, page 466

08 0672318350 CH05 05•02•2003 2:37 PM Page 442

this, set the parameter `normalize` to `TRUE`; otherwise, make it `FALSE`. This helps in many lighting and back face removal calculations to know that the normal to a polygon or plane has length 1.0.

Example:

```
VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// create the plane
PLANE3D_Init(&plane, &p, &n, TRUE);
```

Function Prototype:

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane);
```

Functional Listing:

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane)
{
// test if the point in on the plane, in the positive halfspace
// or negative halfspace
float hs = plane->n.x*(pt->x - plane->p0.x) +
      plane->n.y*(pt->y - plane->p0.y) +
      plane->n.z*(pt->z - plane->p0.z);

// return half space value
return(hs);

} // end Compute_Point_In_Plane3D
```

Purpose:

`float Compute_Point_In_Plane3D()` is a rather interesting function. It computes the half space that the test point lies in. In other words, many times we want to determine whether something is on one side of a plane or the other (or in the plane). That's what this function does. Figure 5.21 illustrates the logic. The function returns `0.0` if the point lies in the plane, a positive number if the point lies in the positive half space (that is, the half space the normal points to), and the function returns a negative number if the point lies in the negative half space. Check out `DEMOII5_2.CPP¦EXE`; it allows you to define a plane and then send test points to it to determine the half space.

Playrix Ex. 1023, page 467

08 0672318350 CH05 05•02•2003 2:37 PM Page 443

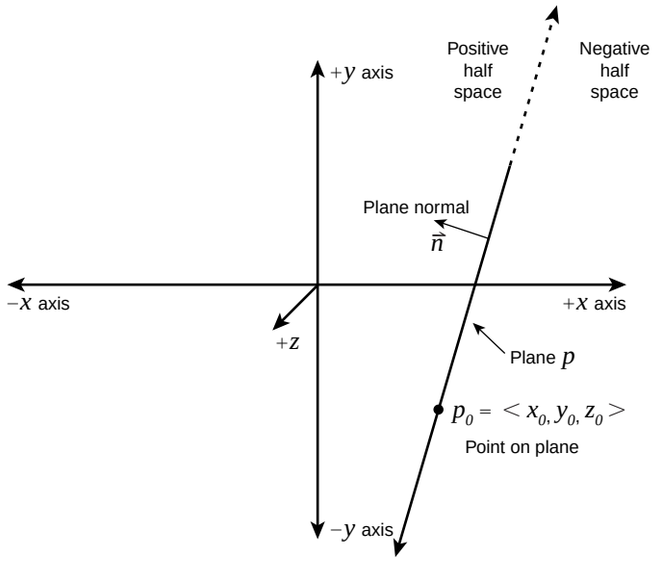


FIGURE 5.21 The half-space logic codes.

Example:

```
VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// create the plane
PLANE3D_Init(&plane, &p, &n, TRUE);

// this should be in the positive half space
POINT3D p_test = {50,50,50};

// test the point
float hs = Compute_Point_In_Plane3D(&p_test, &plane);
```

Function Prototype:

```
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline, PLANE3D_PTR plane,
                    float *t, POINT3D_PTR pt);
```

Playrix Ex. 1023, page 468

08 0672318350 CH05 05•02•2003 2:37 PM Page 444

Functional Listing:

```
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline,
                    PLANE3D_PTR plane,
                    float *t, POINT3D_PTR pt)
{
// this function determines where the sent parametric line
// intersects the plane the function will project the line
// infinitely in both directions, to compute the intersection,
// but the line segment defined by p intersected the plane iff t e [0,1]
// also the function returns 0 for no intersection, 1 for
// intersection of the segment and the plane and 2 for intersection
// of the line along the segment and the plane 3, the line lies
// in the plane

// first test of the line and the plane are parallel, if so
// then they can't intersect unless the line lies in the plane!

float plane_dot_line = VECTOR3D_Dot(&pline->v, &plane->n);

if (fabs(plane_dot_line) <= EPSILON_E5)
  {
  // the line and plane are co-planer, does the line lie
  // in the plane?
  if (fabs(Compute_Point_In_Plane3D(&pline->p0, plane)) <= EPSILON_E5)
   return(PARM_LINE_INTERSECT_EVERYWHERE);
  else
   return(PARM_LINE_NO_INTERSECT);
  } // end if

// from chapter 4 we know that we can solve for the t where
// intersection occurs by
// a*(x0+vx*t) + b*(y0+vy*t) + c*(z0+vz*t) + d =0
// t = -(a*x0 + b*y0 + c*z0 + d)/(a*vx + b*vy + c*vz)
// x0,y0,z0, vx,vy,vz, define the line
// d = (-a*xp0 - b*yp0 - c*zp0), xp0, yp0, zp0, define the point on the plane

*t = -(plane->n.x*pline->p0.x +
    plane->n.y*pline->p0.y +
    plane->n.z*pline->p0.z -
    plane->n.x*plane->p0.x -
    plane->n.y*plane->p0.y -
    plane->n.z*plane->p0.z) / (plane_dot_line);
```

Playrix Ex. 1023, page 469

08 0672318350 CH05 05•02•2003 2:37 PM Page 445


```
// now plug t into the parametric line and solve for x,y,z
pt->x = pline->p0.x + pline->v.x*(*t);
pt->y = pline->p0.y + pline->v.y*(*t);
pt->z = pline->p0.z + pline->v.z*(*t);

// test interval of t
if (*t>=0.0 && *t<=1.0)
  return(PARM_LINE_INTERSECT_IN_SEGMENT );
else
  return(PARM_LINE_INTERSECT_OUT_SEGMENT);

} // end Intersect_Parm_Line3D_Plane3D
```

Purpose:

`int Intersect_Parm_Line3D_Plane3D()` computes the intersection of a 3D parametric line and a 3D plane. The parametric value point of intersection is returned in `t` as well as the actual point of intersection in `pt`. However, before these data are used, the return value of `Intersect_Parm_Line3D_Plane3D()` can be tested to determine the details of the intersection. The values are the same as with the 2D intersection function, with the addition of a flag that indicates that the line lies in the plane itself. Here are the flags again for reference:

```
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

Also take a look at `DEMOII5_3.CPP¦EXE` on the CD; it allows you to define a 3D plane and parametric line and then computes their intersection.

Example:

```
POINT3D p1 = {5,5,-5}, p2 = {5,5,5},pt;
PARMLINE3D pl;
float t;

// create a parametric line from p1 to p2
// note this line is parallel to the z-axis
Init_Parm_Line3D(&p1, &p2, &pl);

VECTOR3D n={0,0,1};
POINT3D p={0,0,0};
PLANE3D plane;
```


Playrix Ex. 1023, page 470

08 0672318350 CH05 05•02•2003 2:37 PM Page 446


```
// create the plane, note this plane
// is the x-y plane
PLANE3D_Init(&plane, &p, &n, TRUE);

// compute the point of intersection
// note it should be (5,5,0)
int intersection_type =
   Intersect_Parm_Line3D_Plane3D(&pl, &plane, &t, &pt);
```

Quaternion Support Functions

I have to tell you—writing the quaternion functions wasn't easy! The problem is making sure that they all work. To double-check everything, you have to do it on paper, and because quaternions don't really have a whole lot of physical meaning, you can't do those nice graph paper sanity checks. However, I think I finally have them all working. The only thing I left out is a couple matrix-quaternion conversion functions and spherical linear interpolation (SLERP). We'll add those when we need them, most probably in the camera work and or animation sections of the book. Nonetheless, I really like quaternions because they make many operations so simple. For example, rotation about an arbitrary line is trivial with quaternions, whereas it's a pain with Euler rotations. Anyway, to refresh your memory, a quaternion is a hyper complex number with four components where one of the components is the real part (usually referred to as q0 or w) and the other three components are the imaginary part (usually referred to as q1, q2, q3). Moreover, quaternions are usually written in one of the following ways:

$\mathbf{q} = q_0 + q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$

or

$\mathbf{q} = q_0 + \mathbf{q}_v$, where $\mathbf{q}_v = q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$

So, we basically need to keep track of four floating-point numbers, and we should be able to access them in a number of ways. The data structure `QUAT` in the following code implements this functionality with a number of unions:

```
typedef struct QUAT_TYP
{
union
  {
  float M[4]; // array indexed storage w,x,y,z order

  // vector part, real part format
  struct
     {
     float  q0; // the real part
```

Playrix Ex. 1023, page 471

08 0672318350 CH05 05•02•2003 2:37 PM Page 447


```
    VECTOR3D qv; // the imaginary part xi+yj+zk
    };
  struct
    {
    float w,x,y,z;
    };
  }; // end union

} QUAT, *QUAT_PTR;
```

With this structure, we can access the quaternion as an array, as a real and a vector, and as four unique identifiers. This makes writing various algorithms a snap! Anyway, here are the quaternion functions:

Function Prototypes:

```
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta);

void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v, float theta);
```


Functional Listing:

```
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta)
{
// initializes a quaternion based on a 3d direction vector and angle
// note the direction vector must be a unit vector
// and the angle is in rads

float theta_div_2 = (0.5)*theta; // compute theta/2

// compute the quaternion, note this is from chapter 4
// pre-compute to save time
float sinf_theta = sinf(theta_div_2);

q->x = sinf_theta * v->x;
q->y = sinf_theta * v->y;
q->z = sinf_theta * v->z;
q->w = cosf( theta_div_2 );

} // end VECTOR3D_Theta_To_QUAT
```

Purpose:

`void VECTOR*D_Theta_To_QUAT()` creates a rotation quaternion based on a direction vector, `v`, and an angle, `theta`. Figure 5.22 illustrates the relationship. This function is

Playrix Ex. 1023, page 472

08 0672318350 CH05 05•02•2003 2:37 PM Page 448

primarily used to create quaternions for rotations of points. Note that the direction vector `v` must be a unit vector. Also, the 4D vector version disregards the `w` component of the vector.

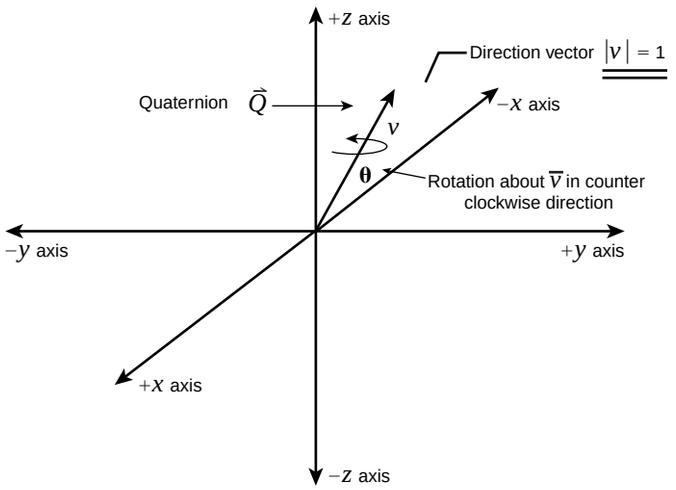


FIGURE 5.22 The rotation quaternion construction.

Example:

```
// create the vector to rotate about
// in this case the diagonal of the unit cube
// for octant 1
VECTOR3D v={1,1,1};
QUAT qr;

// normalize v
VECTOR3D_Normalize(&v);

float theta = DEG_TO_RAD(100); // 100 degrees

// create a rotation quaternion about v
// with angle theta
VECTOR3D_Theta_To_QUAT(&q, &v,theta);
```

Function Prototype:

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float
theta_y, float theta_x);
```

Playrix Ex. 1023, page 473

08 0672318350 CH05 05•02•2003 2:37 PM Page 449

Functional Listing:

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float theta_y, float theta_x)
{
// this function initializes a quaternion based on the zyx
// multiplication order of the angles that are parallel to the
// zyx axis respectively, note there are 11 other possibilities
// this is just one, later we may make a general version of the
// the function

// precompute values
float cos_z_2 = 0.5*cosf(theta_z);
float cos_y_2 = 0.5*cosf(theta_y);
float cos_x_2 = 0.5*cosf(theta_x);

float sin_z_2 = 0.5*sinf(theta_z);
float sin_y_2 = 0.5*sinf(theta_y);
float sin_x_2 = 0.5*sinf(theta_x);

// and now compute quaternion
q->w = cos_z_2*cos_y_2*cos_x_2 + sin_z_2*sin_y_2*sin_x_2;
q->x = cos_z_2*cos_y_2*sin_x_2 - sin_z_2*sin_y_2*cos_x_2;
q->y = cos_z_2*sin_y_2*cos_x_2 + sin_z_2*cos_y_2*sin_x_2;
q->z = sin_z_2*cos_y_2*cos_x_2 - cos_z_2*sin_y_2*sin_x_2;

} // EulerZYX_To_QUAT
```

Purpose:

`void EulerZYX_To_QUAT()` creates a rotation quaternion based on the Euler angles parallel to the z, y, and x axes, respectively. This is your basic camera transform. Of course, there are a total of six ways (permutations = 3!) to multiply x*y*z together, but this is the most common. Use this function to convert your Euler rotation angles to a quaternion.

Example:

```
QUAT qzyx;

// create rotation angles
float theta_x = DEG_TO_RAD(20);
float theta_y = DEG_TO_RAd(30);
float theta_z = DEG_TO_RAD(45);

// create rotation quaternion
EulerZYX_To_QUAT(&qzyx,theta_z,theta_y,theta_x);
```

Playrix Ex. 1023, page 474

08 0672318350 CH05 05•02•2003 2:37 PM Page 450

Function Prototype:

```
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta);
```

Functional Listing:

```
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta)
{
// this function converts a unit quaternion into a unit direction
// vector and rotation angle about that vector

// extract theta
*theta = acosf(q->w);

// pre-compute to save time
float sinf_theta_inv = 1.0/sinf(*theta);

// now the vector
v->x  = q->x*sinf_theta_inv;
v->y  = q->y*sinf_theta_inv;
v->z  = q->z*sinf_theta_inv;

// multiply by 2
*theta*=2;

} // end QUAT_To_VECTOR3D_Theta
```

Purpose:

`void QUAT_To_VECTOR3D_Theta()` converts a unit rotation quaternion into a unit 3D vector and a rotation angle theta about that vector. This function is basically the opposite of `VECTOR*D_Theta_To_QUAT()`.

Example:

```
QUAT q;
// assume q now has a unit rotation quaternion in it

// storage for the vector and angle
float theta;
VECTOR3D v;

// now convert the quat to a vector and an angle
QUAT_To_VECTOR3D_Theta(&q, &v, &theta);
```

Playrix Ex. 1023, page 475

08 0672318350 CH05 05•02•2003 2:37 PM Page 451

Function Prototype:

```
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
```

Purpose:

`void QUAT_Add()` adds the quaternions `q1` and `q2` and stores the sum in `qsum`.

Example:

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qsum;

// add em
QUAT_Add(&q1, &q2, &qsum);
```

Function Prototype:

```
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
```

Purpose:

`void QUAT_Sub()` subtracts the quaternion `q2` from `q1` and stores the difference in `qdiff`.

Example:

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qdiff;

// subtract em
QUAT_Sub(&q1, &q2, &qdiff);
```

Function Prototype:

```
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
```

Purpose:

`void QUAT_Conjugate()` computes the conjugate of the quaternion `q` and returns in `qconj`.

Example:

```
QUAT q = {1,2,3,4}, qconj;

// compute conjugate
QUAT_Conjugate(&q, &qconj);
```

Function Prototype:

```
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
```

Playrix Ex. 1023, page 476

08 0672318350 CH05 05•02•2003 2:37 PM Page 452

Purpose:

`void QUAT_Scale()` scales the quaternion `q` by the factor `scale` and stores the result in `qs`.

Example:

```
QUAT q = {1,2,3,4}, qs;

// scale q by 2
QUAT_Scale(&q, 2, &qs);
```

Function Prototype:

```
void QUAT_Scale(QUAT_PTR q, float scale);
```

Purpose:

`void QUAT_Scale()` scales the quaternion `q` by the factor `scale` in place; that is, it modifies `q` directly.

Example:

```
QUAT q = {1,2,3,4};

// scale q by 2
QUAT_Scale(&q, 2);
```

Function Prototype:

```
float QUAT_Norm(QUAT_PTR q);
```

Purpose:

`float QUAT_Norm(QUAT_PTR q)` returns the norm of the quaternion `q`; that is, its length.

Example:

```
QUAT q = {1,2,3,4};

// whats the length of q?
float qnorm = QUAT_Norm(&q);
```

Function Prototype:

```
float QUAT_Norm2(QUAT_PTR q);
```

Purpose:

`float QUAT_Norm2(QUAT_PTR q)` returns the norm squared of the quaternion `q`; that is, its length squared. This function is useful because many times we need the norm of a

Playrix Ex. 1023, page 477

08 0672318350 CH05 05•02•2003 2:37 PM Page 453

quaternion squared. Thus, calling `QUAT_Norm2()` rather than calling `QUAT_Norm()` and then squaring the return value saves us both a `sqrt()` call and a multiplication.

Example:

```
QUAT q = {1,2,3,4};

// whats the length of q*q?
float qnorm2 = QUAT_Norm2(&q);
```

Function Prototype:

```
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
```

Purpose:

`void QUAT_Normalize()` normalizes the quaternion `q` and sends the result back in `qn`.

> **TIP**
>
> Remember, all rotation quaternions must be unit quaternions.

Example:

```
QUAT q = {1,2,3,4}, qn;

// normalize q
QUAT_Normalize(&q, &qn);
```

Function Prototype:

```
void QUAT_Normalize(QUAT_PTR q);
```

Purpose:

`void QUAT_Normalize()` normalizes the quaternion `q` in place; that is, it modifies `q` itself.

Example:

```
QUAT q = {1,2,3,4};

// normalize q in place
QUAT_Normalize(&q);
```

Function Prototype:

```
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

Playrix Ex. 1023, page 478

08 0672318350 CH05 05•02•2003 2:37 PM Page 454

Purpose:

`void QUAT_Unit_Inverse()` computes the inverse of the quaternion `q` and returns the result in `qi`. However, `q` must be a unit quaternion for the function to work because the function is based on the fact that the inverse of a unit quaternion is its conjugate.

Example:

```
QUAT q = {1,2,3,4}, qi;

// normalize q first
QUAT_Normalize(&q);

// now compute inverse
QUAT_Unit_Inverse(&q, &qi);
```

Function Prototype:

```
void QUAT_Unit_Inverse(QUAT_PTR q);
```

Purpose:

`void QUAT_Unit_Inverse()` computes the inverse of the quaternion `q` in place, modifying `q`. However, `q` must be a unit quaternion for the function to work because the function is based on the fact that the inverse of a unit quaternion is its conjugate.

Example:

```
QUAT q = {1,2,3,4};

// normalize q first
QUAT_Normalize(&q);

// now compute inverse
QUAT_Unit_Inverse(&q);
```

Function Prototype:

```
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

Purpose:

`void QUAT_Inverse()` computes the inverse of a general non-unit quaternion `q` and returns the result in `qi`.

Example:

```
QUAT q = {1,2,3,4}, qi;

// now compute inverse
QUAT_Inverse(&q, &qi);
```

Playrix Ex. 1023, page 479

08 0672318350 CH05 05•02•2003 2:37 PM Page 455

Function Prototype:

```
void QUAT_Inverse(QUAT_PTR q);
```

Purpose:

`void QUAT_Unit_Inverse()` computes the inverse of a general non-unit quaternion `q` in place, modifying `q`.

Example:

```
QUAT q = {1,2,3,4};

// now compute inverse
QUAT_Inverse(&q);
```

Function Prototype:

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
```

Functional Listing:

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod)
{
// this function multiplies two quaternions

// this is the brute force method
//qprod->w = q1->w*q2->w - q1->x*q2->x - q1->y*q2->y - q1->z*q2->z;
//qprod->x = q1->w*q2->x + q1->x*q2->w + q1->y*q2->z - q1->z*q2->y;
//qprod->y = q1->w*q2->y - q1->x*q2->z + q1->y*q2->w - q1->z*q2->x;
//qprod->z = q1->w*q2->z + q1->x*q2->y - q1->y*q2->x + q1->z*q2->w;

// this method was arrived at basically by trying to factor the above
// expression to reduce the # of multiplies

float prd_0 = (q1->z - q1->y) * (q2->y - q2->z);
float prd_1 = (q1->w + q1->x) * (q2->w + q2->x);
float prd_2 = (q1->w - q1->x) * (q2->y + q2->z);
float prd_3 = (q1->y + q1->z) * (q2->w - q2->x);
float prd_4 = (q1->z - q1->x) * (q2->x - q2->y);
float prd_5 = (q1->z + q1->x) * (q2->x + q2->y);
float prd_6 = (q1->w + q1->y) * (q2->w - q2->z);
float prd_7 = (q1->w - q1->y) * (q2->w + q2->z);

float prd_8 = prd_5 + prd_6 + prd_7;
float prd_9 = 0.5 * (prd_4 + prd_8);

// and finally build up the result with the temporary products
```

Playrix Ex. 1023, page 480

08 0672318350 CH05 05•02•2003 2:37 PM Page 456


```
qprod->w = prd_0 + prd_9 - prd_5;
qprod->x = prd_1 + prd_9 - prd_8;
qprod->y = prd_2 + prd_9 - prd_7;
qprod->z = prd_3 + prd_9 - prd_6;

} // end QUAT_Mul
```

Notice that at first I was using the brute-force method of multiplying the quaternions by the definition of multiplication (16 multiplications, 12 additions); and then with some algebra, I simplified the multiplication to 9 multiplications and 27 additions. Normally, you might think this is better, but that might not be the case on floating-point processors.

Purpose:

`void QUAT_Mul()` multiplies the quaternion q1*q2 and stores the result in `qprod`.

Example:

```
QUAT q1={1,2,3,4}, q2={5,6,7,8}, qprod;

// multiply q1*q2
QUAT_Mul(&q1, &q2, qprod);
```

CAUTION

The product `q1*q2` is not equal to `q2*q1` unless `q1` or `q2` is the multiplicative identity. Hence, multiplication is non-commutative for quaternions in general.

Function Prototype:

```
void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR q3, QUAT_PTR qprod);
```

Purpose:

`void QUAT_Triple_Product()` multiplies the three quaternions `q1*q2*q3` and stores the result in `qprod`. This function is useful for rotating points because the transform ($\mathbf{q}^*$)*($\mathbf{v}$)*($\mathbf{q}$) and ($\mathbf{q}$)*($\mathbf{v}$)*($\mathbf{q}^*$) are triple products that rotate vectors or points.

Example:

```
// let rotate the point (5,0,0)
// around the z-axis 45 degrees

// Step 1: create the rotation quaternion
VECTOR3D vz = {0,0,1};
```

Playrix Ex. 1023, page 481

08 0672318350 CH05 05•02•2003 2:37 PM Page 457


```
QUAT qr, // this will hold the rotation quaternion
   qrc; // this will hold its conjugate

// create the rotation quaternion
VECTOR3D_Theta_To_QUAT(&qr, &vz, DEG_TO_RAD(45));

// now its conjugate
QUAT_Conjugate(&qr, &qrc);

// now create a the point to rotate with the q0
// element equal to 0 and the x,y,z as the point
QUAT qp={0,5,0,0};

// now we do the rotation which will rotate
// p about the z-axis 45 degrees, of course
// the rotation axis could have been anything
QUAT_Triple_Product(&qr, &qp, &qrc, &qprod);

// now the result will still have q0=0,
// so we can just extract the point from the
// x,y,z elements of the quaternion
```


Function Prototype:

```
void QUAT_Print(QUAT_PTR q, char *name);
```

Purpose:

`void QUAT_Print()` prints out a quaternion with the given name to the error file.

Example:

```
// set error file to screen
Open_Error_File("", stdout);

QUAT q={1,2,3,4};

QUAT_Print(&q);

// close the error file
Close_Error_File();
```

As a demo of some of the quaternion functions, check out `DEMOII5_4.CPP¦EXE`. It enables you to enter a pair of quaternions and a point in 3D space, and then perform various operations on them and see the results.

Playrix Ex. 1023, page 482

08 0672318350 CH05 05•02•2003 2:37 PM Page 458

Fixed-Point Math Support Functions

Although we didn't cover fixed-point math in the previous chapter, I felt the desire to add some functions to work with it just for the heck of it. If you're rusty on the topic, I refer you to the first *Tricks* for a more detailed discussion. Additionally, on Pentium+ processors, the floating-point unit(s) is as fast or faster than the integer unit(s), thus using fixed-point isn't needed to perform 3D math anymore. However, there are many times that you can write a really optimal algorithm that uses both the floating-point and integer units at the same time, so knowing how to represent and work with fixed-point is still a good thing.

In general, fixed-point is an old trick used on slower processors to perform multiplication and division using integers, but have decimal precision. The trick is using a "fixed point" that's artificially placed somewhere in the binary representation of a number, therefore breaking a number into both a whole part and a decimal part. For the representation for this iteration of the engine, I keep the format first created in the first *Tricks* of 16.16, meaning 16 bits of whole part and 16 bits of decimal part. That means that numbers can be represented that are signed in the magnitude range of +-32768, and a decimal accuracy of $1/2^{16}$.

Now the question is, how to represent a fixed-point number? The answer is just with a standard `int` or `long` data type (if you needed 64 bits, you would use `long`). So, the types in `T3DLIB4.H` define this for us:

```
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

And in `T3DLIB1.H`, there are some legacy constants:

```
// fixed point mathematics constants
#define FIXP16_SHIFT   16
#define FIXP16_MAG    65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

which just simplify working with fixed-point numbers.

Creating a Fixed-Point Number from an Integer

To create a fixed-point number from an integer, all you have do is shift the integer `FIXP16_SHIFT` times to the left, which places the integer in the whole part section of the fixed-point number. But remember not to overflow the fixed-point number. Here's an example:

```
FIXP16 fp1 = (100 << FIXP16);
```

And, of course, there's a macro called `INT_TO_FIXP16()` for this in the header file that does the same thing.

Playrix Ex. 1023, page 483

08 0672318350 CH05 05•02•2003 2:37 PM Page 459

Creating a Fixed Point Number From a Float

Creating a fixed-point number from a float is a little trickier because the binary representation of an integer and a float are fairly alien to each other. So, what we need to do is scale the float by the equivalent magnitude that we shift the integers by. That is, we shift an `int` by 16 places to convert it to a fixed-point number. What we're really doing is multiplying the `int` by $2^{16} = 65536$. So, if we simply multiply the float we want to convert by 65536.0 and then convert it to an `int`, that will do the trick. Here's an example:

```
FIXP16 fp1 = (int)(100.5*65536.0);
```

Notice that we don't shift because the multiplication does that for us, in a manner of speaking. Of course, there's a macro for this called *`FLOAT_TO_FIXP16()`*, which is also in the header and performs the same function. However, you might want to round the number because part of it will be truncated during the conversion to an `int`. Hence, the addition of 0.5 before the conversion helps with loss of data and the final macro does this, which performs something like this:

```
FIXP16 fp1 = (int)(100.5*65536.0+0.5);
```


Converting Back to Floating-Point

If you do want to convert back to floating point, you simply need to extract the whole part of the fixed-point number, which isn't hard to do because it's in the upper 16 bits and it's in normal binary integer format. However, getting the decimal part out is a bit tricky. What you need to do is divide the decimal part by the magnitude it was scaled by originally, which is 65536. However, if you don't want to extract both the whole part and decimal part as separate entities, you can covert a fixed-point number to a float with the following transform:

```
float ((float)fp)/65536.0;
```

which is coded in the macro `FIXP16_TO_FLOAT()`.

Now, as a quick refresher, given two fixed-point numbers, here are the basic operations and how they have to be carried out.

Addition/Subtraction To add or subtract two numbers that are already in fixed-point format, you simply add or subtract them as you normally would. For example:

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

FIXP16 fpsum = fp1 + fp2;
FIXP16 fpdiff = fp1 - fp2;
```

Playrix Ex. 1023, page 484

08 0672318350 CH05 05•02•2003 2:37 PM Page 460

As a more illustrative example, look at this:

```
int x=50, y=23;

// convert x,y to fixed point 16.16 format
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
FIXP16 fsum;

// now lets add them
fsum = fpx + fpy;
```

which simplifies to

```
fsum = x*65536 + y*65536 = (x+y)*65536;
```

Therefore, the result does indeed sum (x+y) and has the proper scale of 65536. The proof for subtraction is similar.

Multiplication Multiplication of fixed-point numbers is a little tricky. The reason why is that each number is scaled by a factor. In the case of 16.16 format, each fixed-point number is scaled by 65536. Thus, if we multiply two fixed-point numbers, we need to make sure that the result is scaled by only 65536 and not $(65536)^2$! The following example illustrates the problem by using a numeric example:

```
// numbers we are going to convert to fixed point
int x=50, y=23;

// convert x,y to fixed point 16.16 format
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

Now, the problem occurs when we try to multiply (`fpx*fpy`). If we look at the numeric results, we get this:

```
fpx*fpy = (x*65536) * (y*65536) = (x*y)*(65536*65536);
```

See the problem? The answer is scaled by 65536^2, rather than 65536. The solution is to simply scale the result back. However, the problem with that is that the act of multiplying the two fixed-point numbers (and then scaling after the fact) actually overflows a 32-bit number. Therefore, our only options are to use 64-bit math (which we'll see momentarily) or to pre-scale down both the multiplier and the multiplicand before the multiplication so that the resulting product is scaled by only 65536. Here's one way of doing it:

```
 = (fpx/256)*(fpy/256)
```

Playrix Ex. 1023, page 485

08 0672318350 CH05 05•02•2003 2:37 PM Page 461

Substituting x,y:

```
= ((x*65536)/256) * ((y*65536)/256)
= (x*256)*(y*256) = (x*y)*65536
```

This is correct, but the problem with this approach is that during the scale down, we lose 8 bits of accuracy of both the multiplier and multiplicand, which isn't acceptable because that's the whole point of fixed-point math: to keep the decimal point accuracy. However, we can save the day with 64-bit math using assembly language.

Division Division has the same problem as multiplication does, but it's in reverse. The problem is when one fixed-point number is divided by another, the decimal portion is lost. That is, the scale of the result is 1.0, rather than 65536. Here's an example to illustrate this:

```
// numbers we are going to convert to fixed point
int x=50, y=23;

// convert x,y to fixed point 16.16 format
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```


Now, let's divide and see what happens:

```
fpx/fpy = (x*65536) / (y*65536) = (x/y);
```

That's definitely not good. Again, we have a dilemma and there are two ways to solve it. We can use 64-bit division in assembly language, or we can pre-scale the numerator and denominator so that the results of the division are scaled by 65536. But again, we're trapped to the constraints of 32-bit math if we use the type `int`. Scaling the numerator up means losing accuracy on the whole part, and scaling the denominator down means losing accuracy on the decimal part, but that's our only choice. So, we can scale up the numerator by 256 (8-bits) and scale down the denominator by 256 (8-bits), and the result will be properly scaled by 65536. However, during the scaling transform, we turn the numerator from 16.16 into 8.16 and the denominator into 16.8, which means that if we want to handle signed numbers, the numerator can't be larger than 255 and the denominator can't be smaller than $1/2^8 \approx 0.0039$.

Here's our example fixed to work:

```
// numbers we are going to convert to fixed point
int x=50, y=23;

// convert x,y to fixed point 16.16 format
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

Playrix Ex. 1023, page 486

08 0672318350 CH05 05•02•2003 2:37 PM Page 462

Now, let's divide with the scaling factors and see what happens:

```
= (x*256*65536) / (y*65536/256)
= (x*256*256/y)*65536/65536
= (x/y)*(256*256) = (x/y)*65536
```

TIP

These examples illustrate that there's no rule that states all fixed-point numbers have to be 16.16; some of your numbers may be 16.16, some 24.8, and still others 0.32. The point is that as long as you keep track of the decimal and overflow and underflow, the math will work out.

With all that in mind, let's take a look at the functions along with their assembly language internals.

Function Prototype:

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
```

Function Listing:

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2)
{
// this function computes the product fp_prod = fp1*fp2
// using 64 bit math, so as not to loose precision

FIXP16 fp_prod; // return the product

_asm {
   mov eax, fp1   // move into eax fp2
   imul fp2     // multiply fp1*fp2
   shrd eax, edx, 16 // result is in 32:32 format
            // residing at edx:eax
            // shift it into eax alone 16:16
   // result is sitting in eax
   } // end asm

} // end FIXP16_MUL
```

The assembly language is pretty straightforward; the only interesting thing is that the Intel 32-bit processors have support for 64-bit math!

Purpose:

`FIXP16 FIXP16_MUL()` multiplies the two fixed-point numbers `fp1*fp2` using 64-bit math and returns the result. Note that there's no loss of accuracy because 64-bit math is used.

Playrix Ex. 1023, page 487

08 0672318350 CH05 05•02•2003 2:37 PM Page 463

Example:

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

// do the multiply
FIXP16 fpprod = FIXP16_Mul(fp1,fp2);
```

Function Prototype:

```
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2);
```

Function Listing:

```
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2)
{
// this function computes the quotient fp1/fp2 using
// 64 bit math, so as not to loose precision

_asm {
   mov eax, fp1   // move dividend into eax
   cdq        // sign extend it to edx:eax
   shld edx, eax, 16 // now shift 16:16 into position in edx
   sal eax, 16    // and shift eax into position since the
            // shld didn't move it -- DUMB! uPC
   idiv fp2     // do the divide
   // result is sitting in eax
   } // end asm

} // end FIXP16_DIV
```

The division algorithm is a little tricky. The main point is that first the dividend must be sign extended to 64 bits, and the 64-bit shifts to place the dividend properly are a little contrived since the 64-bit double-shift instructions don't shift the source register `eax`—it stays in place. Thus, we need an extra shift to place it after the initial 64-bit shift. Refer to any Intel assembly language manual for more details on these stupid 64-bit shifts that do half a job!

Purpose:

`FIXP16 FIXP16_DIV()` divides `fp1/fp2` and returns the result. Note that the function uses 64-bit math, so there's no loss of accuracy.

Playrix Ex. 1023, page 488

08 0672318350 CH05 05•02•2003 2:37 PM Page 464

Example:

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

// do the divide
FIXP16 fpdiv = FIXP16_Mul(fp1,fp2);
```

> **NOTE**
>
> Although fixed-point math used to blow the doors off floating-point math, as you can see, there are quite a few instructions involved to perform both the multiplication and division. The rule of thumb is this. Fixed-point addition and subtraction are usually the same speed as floating-point; floating-point multiplication and division are, amazingly enough, as fast or faster than fixed-point.

Function Prototype:

```
void FIXP16_Print(FIXP16 fp);
```

Purpose:

`void FIXP16_Print()` prints out a fixed-point number as if it were a floating-point number.

Example:

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16_Print(fp1);
```

Finally, make sure to check out `DEMOII5_5.CPP¦EXE` on the CD. It enables you to enter two floating-point values, converts them to fixed-point, performs all the operations on them, and prints the results so that you can see the accuracy.

Equation Solving Support

This next pair of functions is really cool. They enable you to solve systems of equations of the form **A*X** = **B**. All you need is the coefficient matrix **A** and the constant matrix **B**, and if a solution exists, the functions will find it. For example, given the 3x3 system:

```
3*x + 2*y - 5*z = 6
 x - 3*y + 7*z = -4
5*x + 9*y - 2*z = 5
```

Playrix Ex. 1023, page 489

08 0672318350 CH05 05•02•2003 2:37 PM Page 465

We deduce that

```
     ¦3    2   -5¦
A =  ¦1   -3    7¦
     ¦5    9   -2¦

X = [x y z]t

B = [6 –4 6]t
```

And we can use the transform $\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$. This is exactly what the solvers do, more or less.

Function Prototype:

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X, MATRIX1X2_PTR B);
```

Function Listing Reference:

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X, MATRIX1X2_PTR B)
{
// solves the system AX=B and computes X=A(-1)*B
// by using cramers rule and determinates

// step 1: compute determinate of A
float det_A = Mat_Det_2X2(A);

// test if det(a) is zero, if so then there is no solution
if (fabs(det_A) < EPSILON_E5)
  return(0);

// step 2: create x,y numerator matrices by taking A and
// replacing each column of it with B(transpose) and solve
MATRIX2X2 work_mat; // working matrix

// solve for x //////////////////

// copy A into working matrix
MAT_COPY_2X2(A, &work_mat);

// swap out column 0 (x column)
MAT_COLUMN_SWAP_2X2(&work_mat, 0, B);

// compute determinate of A with B swapped into x column
float det_ABx = Mat_Det_2X2(&work_mat);
```


Playrix Ex. 1023, page 490

08 0672318350 CH05 05•02•2003 2:37 PM Page 466


```
// now solve for X00
X->M00 = det_ABx/det_A;

// solve for y //////////////////

// copy A into working matrix
MAT_COPY_2X2(A, &work_mat);

// swap out column 1 (y column)
MAT_COLUMN_SWAP_2X2(&work_mat, 1, B);

// compute determinate of A with B swapped into y column
float det_ABy = Mat_Det_2X2(&work_mat);

// now solve for X01
X->M01 = det_ABy/det_A;

// return success
return(1);

} // end Solve_2X2_System
```

Purpose:

`int Solve_2X2_System()` solves the system of equations **A*X** = **B**, where **A** is 2×2, **X** is 1×2, and **B** is 1×2. If the solution exists, it is returned in **B** and the function returns a `1`; otherwise, the function returns a `0` and **B** is undefined. Also, take a look at the function listing to see what's going on inside. Basically, Cramer's rule is used to solve the system. `Solve_2X2_System()` is a good example of many matrix functions being used, so pay close attention. Also take a look at `DEMOII5_6.CPP¦EXE` on the CD—it enables you to enter a system of equations and solves it for you!

Example:

`DEMOII5_6.CPP` allows you to input the coefficient matrix **A** along with constant matrix **B**, and then solves the system **A*X** = **B** with Cramer's rule. The following is a code dump of the main calls:

```
void main()
{
MATRIX2X2 mA;
MATRIX1X2 mB;
MATRIX1X2 mX;

// open the error system, so we can see output
```

Playrix Ex. 1023, page 491

08 0672318350 CH05 05•02•2003 2:37 PM Page 467


```
// notice the last parameter "stdout" this tell the error
// system that we don't want to write errors to a text file
// but straight out to the screen!
Open_Error_File("", stdout);

// get matrix A
printf("\nEnter the values for the matrix A (2x2)");
printf("\nin row major form m00, m01, m10, m11?");
scanf("%f, %f, %f, %f", &mA.M00, &mA.M01, &mA.M10, &mA.M11);

printf("\nEnter the values for matrix B (2x1)");
printf("\nin column major form m00, m10?");
scanf("%f, %f", &mB.M00, &mB.M01);

// technically you can't multiply a 2x2 * 1x2,
// but you can multiply is a 2x2 * (1x2)t
// that is the transpose of a 1x2 is a 2x1,
// bottom line is the B matrix is just 2 numbers!

// now solve the system
if (Solve_2X2_System(&mA, &mX, &mB))
  {
  // now print results
  VECTOR2D_Print((VECTOR2D_PTR)&mX, "Solution matrix mX");
  } // end if
else
  printf("\nNo Solution!");

// close the error system
Close_Error_File();

} // end main
```


Function Prototype:

```
int Solve_3X3_System(MATRIX3X3_PTR A, MATRIX1X3_PTR X, MATRIX1X3_PTR B);
```

Purpose:

`int Solve_3X3_System()` solves the system if equations **A*X** = **B** where **A** is 3×3, **X** is 1×3, and **B** is 1×3. If the solution exists, it's returned in **B** and the function returns a `1`; otherwise, the function returns a `0` and **B** is undefined.

Playrix Ex. 1023, page 492

08 0672318350 CH05 05•02•2003 2:37 PM Page 468

Example:

```
MATRIX3X3 mA = {1,2,9, 4,-3,6, 1,0,5};
MATRIX1X3 mB = {1,2,3};
MATRIX1X3 mX;

// solve the system
if(Solve_3X3_System(&mA, &mX, &mB))
  {
  // now print results
  VECTOR3D_Print((VECTOR3D_PTR)&mX, "Solution matrix mX");
  } // end if
else
  printf("\nNo Solution!");
```

Floating-Point Unit Math Primer

The old 80387 and internal math co-processor of PentiumX (FPU unit) are probably some of the most mysterious components of the PC's processing suite. To tell you the truth, it's next to impossible to find any good references on FPU programming other than a couple hard-to-find books and the Intel Instruction set docs. However, that's all going to change right now.

> **NOTE**
>
> These days, new generations of Pentium processors may have more than one FPU, so I'm going to generically refer to floating-point programming and floating-point co-processors as the FPU.

We're going to delve into the FPU at least to the point that we understand what's going on and how to actually program it using its own instruction set. Of course, the C/C++ compiler you're using generates FPU instructions when you compile code that uses floating-point operations. However, this code can be optimized in many cases with assembly language and full 64-bit instructions as our 64-bit multiply and divide illustrated using the integer unit to do fixed-point math.

Now, there's one other gotcha. On the Pentium models with MMX support, the FPU registers are aliased to the MMX registers, meaning that you can't mix MMX code and FPU code without doing a lot of context switching. Now, I'm not going to talk about MMX programming since it's its really added instructions to perform SIMD (single instruction multiple data) processing and really is a topic unto itself and better explained in a optimization book. If you're interested, Intel has a nice *MMX Programming Guide* that you can obtain from the developers section of Intel's site.

So, how does the FPU work? First, from a programming point of view, it doesn't matter whether there's one or more FPUs, whether they're internal or external (high-performance

Playrix Ex. 1023, page 493

08 0672318350 CH05 05•02•2003 2:37 PM Page 469

FPUs), the instruction sets are roughly the same. At least the most common operations are, which is what we're interested in learning.

CAUTION

FPU programming must be done at the assembly language level. But we're going to at least use the inline assembler, so even if you aren't an assembly language guru, learning FPU programming is fairly easy because it's like a little computer with a stack—nothing more.

The FPU Architecture

The FPU is a stack-based machine that has only one mission in life: to compute mathematical floating-point calculations for the CPU core. Figure 5.23 shows the abstract model of the FPU in relationship to the CPU core. Basically, all instructions that the CPU comes across that are meant for the FPU are sent to the FPU where they are executed. Hence, FPU instructions don't directly affect the CPU or its internal registers. Many times the CPU's integer unit and FPU can't work at the same time, but in most cases, the CPU and FPU can work in parallel—especially with U-V pipe architectures like the Pentium. On more advanced models of the Pentium, things get a little weird, so you might want to read up on the Pentium I, II, III, Pro, and all the other flavors, since I gave up trying to keep up with all the rules!

Anyway, FPU can perform many common mathematical operations such as addition, subtraction, multiplication, and division. However, it's also equipped to handle logarithmic, transcendental, numerical conversions, and more. Furthermore, the FPU can actually perform many operations faster than the equivalent integer unit with much higher accuracy. Even in the cases in which the FPU is slower than pure integer operations using the integer unit, it might be worth the extra accuracy—especially in 3D graphics.

Programming the FPU is actually very simple. It has its own instruction set, which we'll cover shortly. Basically, whenever you want a FPU instruction to be executed, you simply place it in your code right with the other assembly language instructions. This brings us to a little problem, and that's that most FPU programming must be done using assembly language and, hence, either external assembly functions or inline (I prefer the inline). However, when I want to write a function that uses the FPU directly, I usually code the entire thing using inline assembly. But of course I do this for little functions that must run very quickly, not 500-line-long, high-level functions.

NOTE

When you set the compiler to high optimization and inline FPU instructions, it does a very good job at emitting FPU code that's nearly optimal.

Playrix Ex. 1023, page 494

08 0672318350 CH05 05•02•2003 2:37 PM Page 470

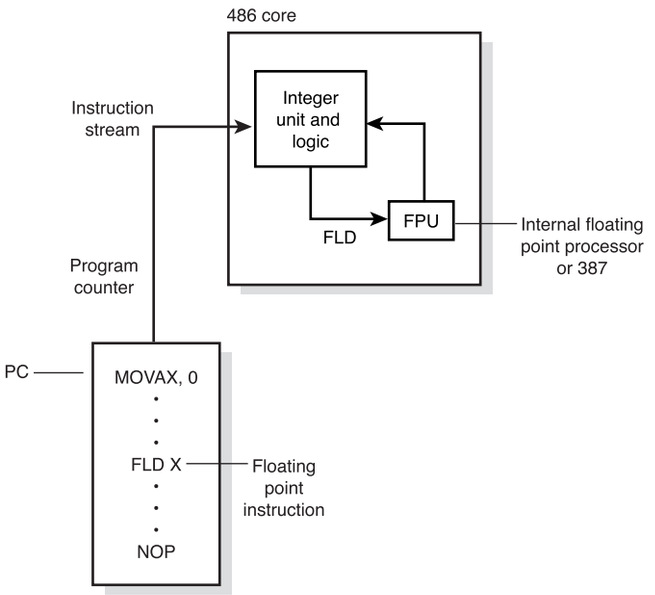


FIGURE 5.23 The floating-point processor performs all floating-point math.

The FPU Stack

Let's talk about the internal stack of the FPU since it's the basis of how everything works. Basically, there are eight stack elements in the FPU and a status word, as shown in Figure 5.24. Each stack element is 80 bits wide with 64 bits for the decimal part, 14 bits for the exponent, and 1 bit for the sign bit.

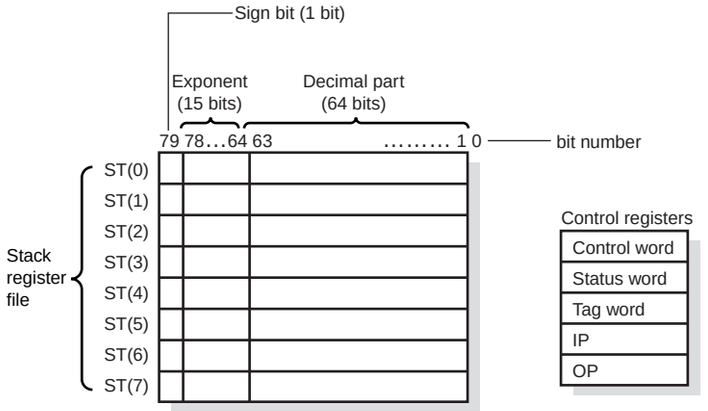


FIGURE 5.24 The FPU core architecture.

Playrix Ex. 1023, page 495

08 0672318350 CH05 05•02•2003 2:37 PM Page 471

However, each stack element can hold different data types, as shown in Figure 5.25. We see that there are short, long, and 10-byte real numbers. The stack elements are used to hold inputs and return outputs; thus, they are the working registers for the FPU, in a sense. We can think of them as a register file, or we can think of them as a stack.

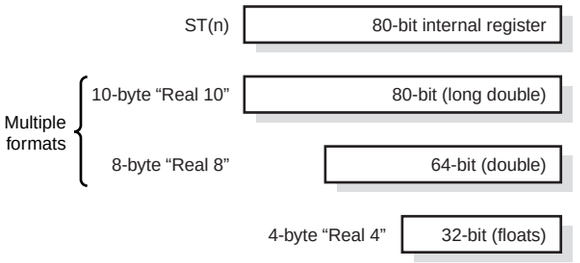


FIGURE 5.25 FPU data formats.

In any case, the stack is referred to using the syntax `ST(n)`, where *n* is the element. For example, take a look at Figure 5.26.

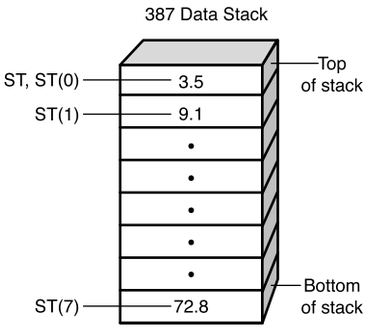


FIGURE 5.26 The FPU stack ordering and access.

Some examples:

`ST` is equivalent to the top of stack (TOS).

`ST(0)` is equivalent to the top of stack.

`ST(1)` is equivalent to the second element of the stack.

`ST(7)` is equivalent to the last element of the stack or element 8.

When programming the FPU, many of the instructions take zero, one, or two operands. This means that sometimes the operand(s) are implied. Also, the operands can either be

Playrix Ex. 1023, page 496

08 0672318350 CH05 05•02•2003 2:37 PM Page 472

constants, memory, or stack elements in most cases, so we have a bit of flexibility in defining what the data is for an operation. Furthermore, many of the instructions place the result on the top of the stack at `ST(0)`. This is a good thing to know if you're expecting some piece of data to remain valid after an operation and it gets obliterated.

Now that we have a bit of an idea of what the FPU does and how it does it, let's take a look at the instruction set and see how to use it.

> **NOTE**
>
> Because there are new Pentium processors popping up all the time, an exhaustive list of FPU instructions is pointless. Instead, Table 5.1 lists the majority of commonly used instructions that you'll probably never use more than 10% of anyway.

The FPU Instruction Set

The FPU is a complete processor with a rather large instruction set. The instructions fall into a few categories, they are

- Data Transfer—Used to move data between CPU and FPU
- Arithmetic—Used to perform standard arithmetic operations
- Transcendental—Used for non-linear operations, such as sin, cos, and so on
- Constants—Used to load commonly used constants such as 1, pi, and so on
- Comparison—Used to compare floating-point numbers
- Control—Used the control and set the state of the FPU

I don't know about you, but I like to see an instruction set first before I learn about it just to get a feel for what's what! Therefore, Table 5.1 lists all the FPU instructions available with a short description of what they do.

TABLE 5.1 The FPU Co-processor Instruction Set

Instruction Name	Description
Data Transfer Instructions	
`FBLD`	Loads a BCD number
`FBSTP`	Stores and pops a BCD number
`FILD`	Loads an integer
`FIST`	Stores an integer
`FISTP`	Stores an integer and pops the stack
`FLD`	Loads a real number
`FSTP`	Stores a real number and pops the stack
`FXCH`	Exchanges two stack elements

Playrix Ex. 1023, page 497

08 0672318350 CH05 05•02•2003 2:37 PM Page 473

TABLE 5.1 Continued

Instruction Name	Description
Arithmetic Instructions	
FABS	Computes the absolute value
FADD	Adds real numbers
FIADD	Adds integers
FADDP	Adds real numbers and pops the stack
FCHS	Changes the sign of a number
FDIV	Divides real numbers
FIDIV	Divides integers
FDIVP	Divides real numbers and pops the stack
FDIVR	Divides real numbers, but reverses which operand is dividend and divisor
FIDIVR	Divides integers, but reverses which operand is dividend and divisor
FDIVRP	Divides real numbers and pops the stack, but reverses which operand is dividend and divisor
FMUL	Multiplies real numbers
FIMUL	Multiplies integers
FMULP	Multiples real numbers and pops the stack
FPREM	Computes partial remainder
FPREM1	Computes partial remainder using IEEE format
FRNDINT	Rounds the operand to an integer
FSCALE	Scales by a power of two
FSUB	Subtracts real numbers
FISUB	Subtracts integers
FSUBP	Subtracts real numbers and pops the stack
FSUBR	Subtracts real numbers, but reverses which operand is subtractor and subtrahend
FISUBR	Subtracts integers, but reverses which operand is subtractor and subtrahend
FSUBRP	Subtracts real numbers and pops the stack, but reverses which operand is subtractor and subtrahend
FSQRT	Computes the square root
FXTRACT	Extracts the exponents and significand from real number
Transcendental Instructions (Note: all angles must be in radians)	
F2XM1	Computes the value (2*x-1)
FCOS	Computes the cosine
FPATAN	Computes the partial arctangent
FPTAN	Computes the partial tangent
FSIN	Computes the sine
FSINCOS	Computes both the sine and cosine
FYL2X	Computes the expression $y*\log_2 x$
FYL2XP1	Computes the expression $y*\log_2 (x+1)$

Playrix Ex. 1023, page 498

08 0672318350 CH05 05•02•2003 2:37 PM Page 474

TABLE 5.1 Continued

Instruction Name	Description
Constants	
FLD1	Loads a 1.0
FLDL2E	Loads $\log_2 e$
FLDL2T	Loads $\log_2 10$
FLDLG2	Loads $\log_{10} 2$
FLDPI	Loads pi
FLDZ	Loads zero
Comparison Operators	
FCOM	Compares reals
FCOMP	Compares reals and pops the stack
FCOMPP	Compares reals and pops the stack twice
FICOM	Compares integers
FICOMP	Compares integers and pops the stack
FTST	Compares the top of stack to zero
FUCOM	Performs unordered compare
FUCOMP	Performs unordered compare along with a pop
FUCOMPP	Performs an unordered compare along with two pops
FXAM	Examine st(0) and place results in condition register
Control Instructions	
FCLEX	Clears all unmasked floating-point exceptions
FNCLEX	Clears all exceptions
FDECSTP	Decrements the stack pointer
FFREE	Clears a stack element making it seem like it was popped
FINCSTP	Increments stack pointer
FINIT	Initializes FPU and checks for exceptions
FNINIT	Initializes FPU without checking for exceptions
FLDCW	Loads the control word
FLDENV	Loads the FPU environment
FNOP	Equivalent to NOP
FRSTOR	Restores the state of the FPU with a given memory area
FSAVE	Saves the state of the FPU into a memory area, but checks for exceptions
FNSAVE	Saves the state of the FPU into a memory area without checking for exceptions
FSTCW	Stores the control word and checks exceptions
FNSTCW	Stores the control word without checking exceptions
FSTENV	Stores the environment and checks exceptions
FNSTENV	Stores the environment without checking exceptions
FSTSW	Stores the status word and checks exceptions
FNSTSW	Stores the status word without checking exceptions
FSTSW AX	Stores the status word into AX and checks exceptions
FNSTSW AX	Stores the status word into AX without checking exceptions
WAIT	Suspends CPU until FPU is complete with operation

Playrix Ex. 1023, page 499

08 0672318350 CH05 05•02•2003 2:37 PM Page 475

Because there are so many instructions, listing an example of each of them would take up too much time; instead, Table 5.2 lists the general format of most instructions. Using this table, you should be able to deduce what the legal operands are for a particular instruction.

TABLE 5.2 The Operand Formats for the FPU

Instruction Format	Syntax Operands	Implied
Classical	F*instruction*	ST,ST(1)
Memory	F*instruction*	ST
	memory_operand	
Register	F*instruction* ST(n), ST	None
	F*instruction* ST,ST(n)	None
Register Pop	F*instruction*P ST(n),ST	None

NOTE

Notice that all the FPU instructions start with a leading F and, in the case of register pop format, the instructions end with a P. Now let's discuss each of the instruction formats, so we know how to use them in programs.

Classical Instruction Format

The classical format treats the FPU's stack as if it were a classic stack. Meaning that all operations refer to the top of stack ST(0) and the second element ST(1). For example, if an addition is requested using the instruction:

```
FADD
```

then the operation that would be performed is ST(0) = ST(0) + ST(1). Therefore, the top of the stack would be replaced by the sum of the topmost two items, as shown in Figure 5.27. In general, classical instructions use ST(0) and ST(1) if two operands are needed, and ST(0) if only one is needed. If the instruction uses two operands, the source topmost operand is typically popped off the stack and the result is placed at the top also.

Memory Instruction Format

The memory instruction format is similar to the classical format in that the stack is still treated like a stack. However, memory format allows memory items to be pushed on the stack and popped off the stack into memory. For example, we might want to load the top of the stack with a memory operand like this:

```
FLD memory
```

Playrix Ex. 1023, page 500

08 0672318350 CH05 05•02•2003 2:37 PM Page 476

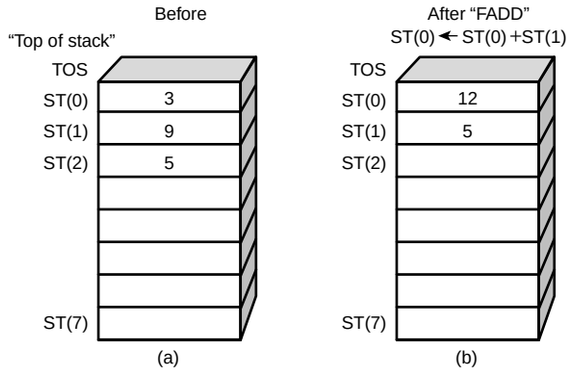


FIGURE 5.27 An example of a classical stack instruction.

And if we wanted to pop the top of stack into a memory location, we could write

```
FSTP memory
```

The results of both of these instructions are shown in Figure 5.28.

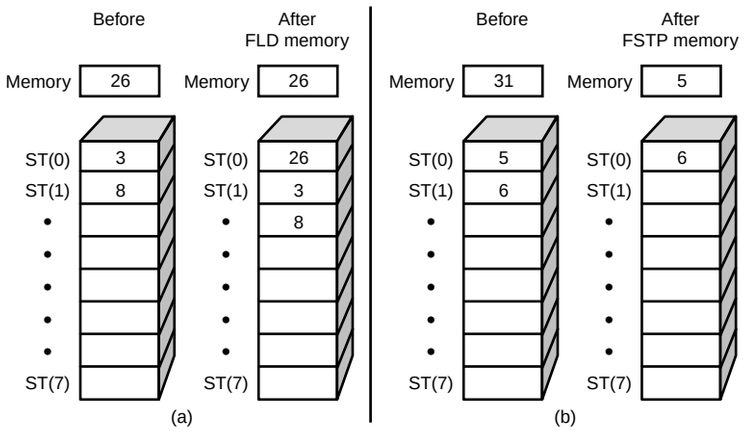


FIGURE 5.28 Examples of memory instruction format.

Register Instruction Format

The register instruction format simply allows the stack elements (registers) to be explicitly referred to. For example, say we want to add `ST(0)` and *`ST(4)`* together. We could write

```
FADD ST(0), ST(4)
```

Playrix Ex. 1023, page 501

08 0672318350 CH05 05•02•2003 2:37 PM Page 477

and the top of stack would hold the sum of `ST(0) + ST(4)`, as is shown in Figure 5.29. Also, when using register instruction format with two operand forms, one of the operands must be the top of stack `ST(0)` or its abbreviation, `ST`.

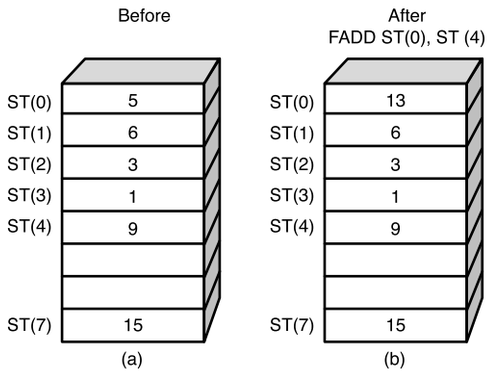


FIGURE 5.29 An example of register instruction format.

Register Pop Instruction Format

The register pop format is identical to the register instruction format except that after the result has been placed into the stack, the stack is popped once.

FPU Examples

Even though we aren't going to look at all the instructions the FPU has to offer, we'll take a look at some of the most representative instructions you'll need. They are

- `FLD`—In all its forms
- `FST`—In all its forms
- `FADD`—In all its forms
- `FSUB`—In all its forms
- `FMUL`—In all its forms
- `FDIV`—In all its forms

Also, the format of each instruction will be

1. Description
2. Instruction syntax and operands (if any)

Playrix Ex. 1023, page 502

08 0672318350 CH05 05•02•2003 2:37 PM Page 478

FLD Examples

The FLD instruction and its various forms is used to load reals (FLD), integers (FILD), and constants such as pi (FLDPI). The FLD instruction has the following formats:

Load a real number:

```
FLD     op1
```

32-bit short real

64-bit long real

80-bit temporary real

ST(n)

Load an integer:

```
FILD    op1
```

16-bit word integer

32-bit short integer

64-bit long integer

Load a constant:

```
FLDcon
```

where *con* is the suffix that determines the constant to load. Refer to the section on constants in Table 5.1.

Let's see some examples now. To load a real 32-bit number from memory, we could write

```
FLD memory32
```

After the execution of the instruction, the top of stack ST(0) would contain the value in memory32. Figure 5.30a shows this.

To load a standard binary 16-bit integer in the FPU, we can write

```
FILD memory16
```

After the instruction executes the top of the stack would hold the 16-bit integer in IEEE format as shown in Figure 5.30b.

NOTE

All integers are converted to reals when processed by the floating-point processor.

Playrix Ex. 1023, page 503

08 0672318350 CH05 05•02•2003 2:37 PM Page 479

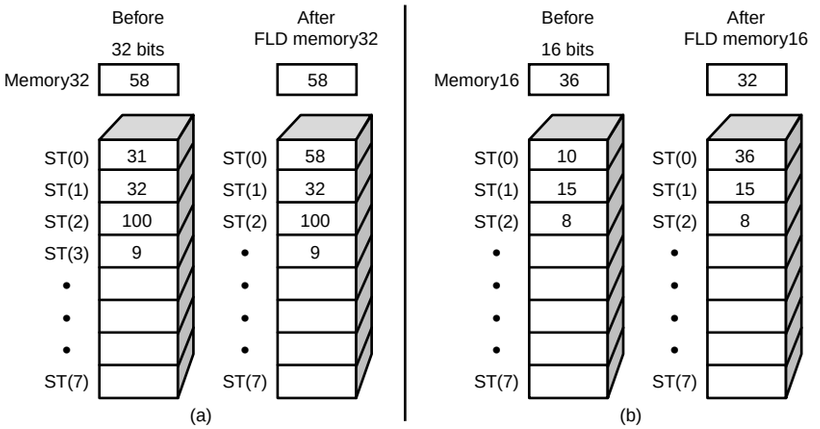


FIGURE 5.30 Examples of the load instruction.

FST Examples

The FST instruction is used to store reals (FST) and integers (FIST). Also, a P can be appended to both instructions, which instructs the FPU to pop the top of stack will after the store is executed. In essence, the top of stack ST(0) is stored into the destination and the stack pointer is adjusted when P is added to the instructions. The FST instruction has the following formats:

Store a real number:

FST *op1*

32-bit short real

64-bit long real

ST(n)

Store a real number and pop the stack:

FSTP *op1*

32-bit short real

64-bit long real

ST(n)

Playrix Ex. 1023, page 504

08 0672318350 CH05 05•02•2003 2:37 PM Page 480

Round the top of stack and store it:

`FIST` *op1*

16-bit word integer

32-bit short integer

64-bit long integer

Round the top of stack, pop, and store it:

`FISTP` *op1*

16-bit word integer

32-bit short integer

64-bit long integer

Also, the top of the stack is always implied to be the source; hence *op1* is the destination. With that in mind, let's see some examples. To store the top of the stack in a standard floating-point location, we could write

```
FST memory32
```

The results of this are shown in Figure 5.31a; notice that the top of stack is not modified. On the other hand, if we were to execute this instruction:

```
FSTP memory32
```

the top of the stack would be popped into `memory32`. Now let's try some integer examples. If we wanted to store the top of the stack into a 32-bit integer, we could write

```
FIST memory32
```

and the value in the top of the stack would be converted to an integer by rounding and stored into `memory32` as shown in Figure 5.31b. `FISTP` works in the same way except the top of stack is popped as shown in Figure 5.31c.

FADD Examples

The `FADD` instruction adds two reals (`FADD`) or adds two integers (`FIADD`). Also, a `P` can be appended to the real version (`FADDP`) that pops the stack after the addition. However, this feature isn't supported for the integer version. The `FADD` instruction has the following forms:

Playrix Ex. 1023, page 505

08 0672318350 CH05 05•02•2003 2:37 PM Page 481

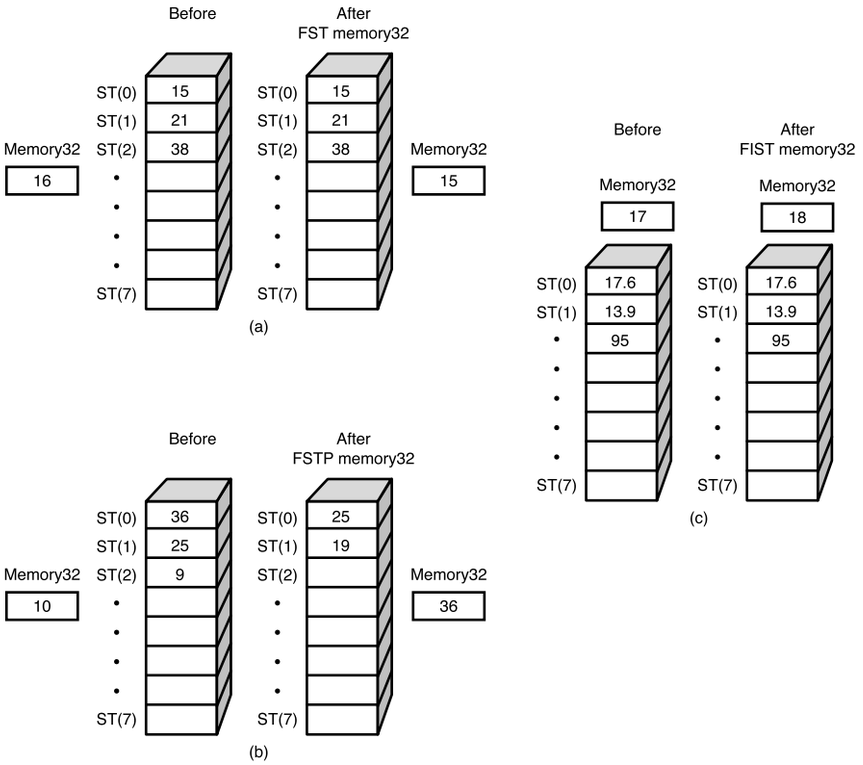


FIGURE 5.31 Examples of the store instructions.

Implied operands are `ST(0)` *and* `ST(1)`*. Also, the stack is popped after the addition.*

```
FADD
```

The top of stack is implied as the destination and one `memory32` *of the operands.*

```
FADD    op1
        memory64
        ST(n)
```

One of the operands must still be the top of stack `ST`*, but the destination need not be.*

```
FADD    op1, op2
        ST(n), ST
        ST, ST(n)
```

Playrix Ex. 1023, page 506

08 0672318350 CH05 05•02•2003 2:37 PM Page 482

The operands are added and the result is popped off the top of stack.

```
FADDP   op1, op2
        ST(n), ST
        ST, ST(n)
```

The integer referred to by *op1* is converted to a real and added to the top of stack.

```
FIADD   op1
        memory16
        memory32
```

Alrighty then! Seems easy enough. Let's try some examples. First, let's add the two top most elements of the stack. The following instructions are equivalent:

```
FADD ST(1)

FADD ST,ST(1)
```

Figure 5.32a shows the results of executing either of the preceding instructions. As you can see, the top of the stack is obliterated with the sum.

However, if we were to use this instruction:

```
FADD ST(2), ST
```

the sum of `ST(0)` and `ST(2)` would be stored into `ST(2)` because it's the destination and `ST(0)` would be unscathed. This is shown in Figure 5.32b. Next, let's try one of the integer forms. Say we wanted to add a 32-bit integer number from memory with the top of the stack `ST(0)`. We could use the following instruction:

```
FIADD memory32
```

The value within `memory32` would be converted to a real and then added to `ST(0)`. The result would be placed back into `ST(0)` as shown in Figure 5.32c. However, if we wanted to pop the top of stack after the addition, we would have to use a real addition because popping isn't supported after integer addition. But we could pop the stack manually with the instruction:

```
FINCSTP
```

Now let's put together a little program that adds two reals and stores the result in a 32-bit integer. Here's the code:

```
float fvalue_1 = 100.2;
float fvalue_2 = 50.5;
int ivalue_1 = 0;
```

Playrix Ex. 1023, page 507

08 0672318350 CH05 05•02•2003 2:37 PM Page 483


```
_asm {
   FLD fvalue_1  // load the first 4 byte real
   FADD fvalue_2  // load and add second 4 byte real
   FISTP ivalue_1 // store the top of stack into the 4
          // byte integer
   } // end asm
```

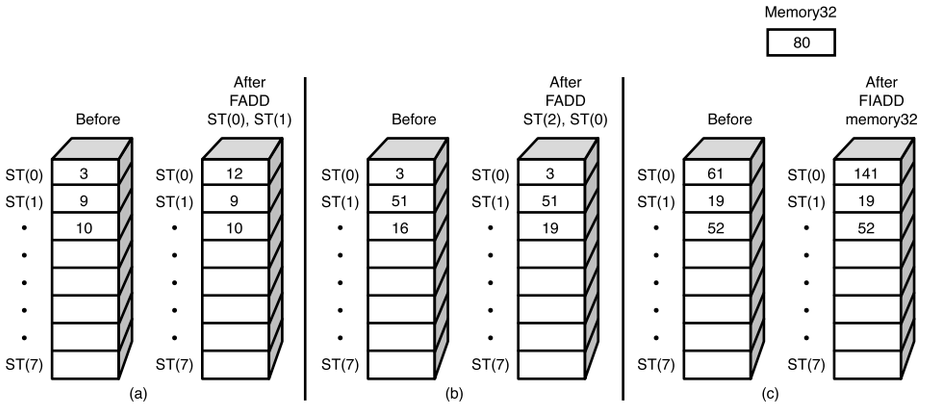


FIGURE 5.32 Examples of the add instructions.

Figure 5.33 shows the program executing step by step.

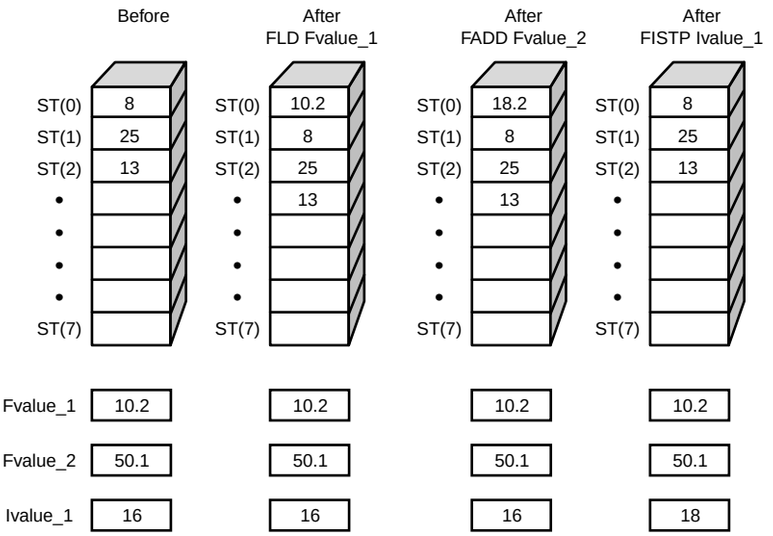


FIGURE 5.33 Execution of a floating-point program and the stack.

Playrix Ex. 1023, page 508

08 0672318350 CH05 05•02•2003 2:37 PM Page 484

FSUB Examples

Implied operand and order are (`ST(1) - ST(0)`). Also, the stack is popped after the subtraction.

```
FSUB
```

The top of stack is implied as the destination and one of the operands; hence, the operation is (`ST(0)` – `op1`).

```
FSUB    op1
        memory32
        memory64
        ST(n)
```

One of the operands must still be the top of stack `ST`, but the destination need not be. Again, the operation performed is (*op1* - *op2*).

```
FSUB    op1, op2
        ST, ST(n)
        ST(n), ST
```

The expression (`op1 - op2`*) is computed and the result is popped off the top of stack.*

```
FSUBP   op1, op2
        ST, ST(n)
        ST(n), ST
```

The integer referred to by `op1` *is converted to a real and subtracted from the top of stack. The result is, of course, placed in* `ST(0)`*.*

```
FISUB   op1
        memory16
        memory32
```

Subtraction works in the same way as addition, except that the source and destination operands play a more important role because (a-b) is different from (b-a). That is, you need to keep an eye on the order in which the subtraction takes place! For example, the following instruction:

```
FSUB memory32
```

subtracts `memory32` from the top of stack `ST(0)`. That is, `ST(0) = ST(0) - memory32`, as is shown in Figure 5.34a. As another example, imagine that we wanted to subtract an integer from the top of the stack `ST(0)` (maybe to compute the decimal part). We could use this instruction:

```
FISUB memory16
```

Playrix Ex. 1023, page 509

08 0672318350 CH05 05•02•2003 2:37 PM Page 485

and the top of stack would have the results `ST(0)` - (real)(`memory16`), as shown in Figure 5.34b. Again, we see that all integers are converted into reals before they are used in calculations.

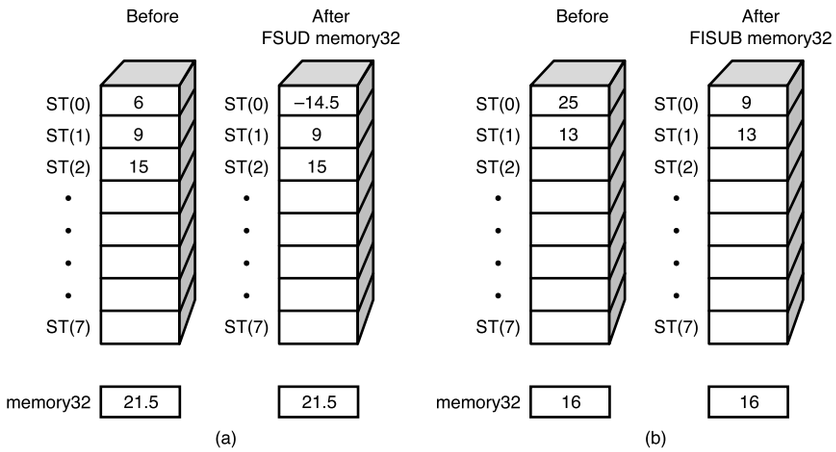


FIGURE 5.34 Examples of the subtract instructions.

FMUL Examples

The `FMUL` instruction is very similar to the `FADD` instruction. It can multiply two reals (`FMUL`), an integer and a real (`FIMUL`), and as usual, a `P` can be appended to the real version of the instruction (`FMULP`), which pops the stack after the multiplication. The *`FMUL`* instruction has the following forms:

*Implied operand are `ST(0)` * `ST(1)`. Also, the stack is popped after the multiplication.*

```
FMUL
```

The top of stack is implied as the destination and one of the operands.

```
FMUL    op1
        memory32
        memory64
        ST(n)
```

One of the operands must still be the top of stack `ST`, but the destination need not be.

```
FMUL    op1, op2
        ST, ST(n)
        ST(n), ST
```

Playrix Ex. 1023, page 510

08 0672318350 CH05 05•02•2003 2:37 PM Page 486

The operands are multiplied and the result is popped off the top of stack.

```
FMULP   op1, op2

        ST, ST(n)

        ST(n), ST
```

The integer referred to by `op1` *is converted to a real and multiplied by the top of stack.*

```
FIMUL   op1

        memory16

        memory32
```

Hmmm ... I'm running out of examples! Let's say that we wanted to do something useful, like compute 5!, which is read as "five factorial" and is equal to 5*4*3*2*1. This little program would do the trick:

```
_asm {
   FLD f_1 // load 1.0 into stack
   FLD f_2 // load 2.0 into stack
   FLD f_3 // load 3.0 into stack
   FLD f_4 // load 4.0 into stack
   FLD f_5 // load 5.0 into stack
   FMUL   // 5*4
   FMUL   // 5*4*3
   FMUL   // 5*4*3*2
   FMUL   // 5*4*3*2*1

    // top of stack ST(0) has 5*4*3*2*1 in it.
    } / /end asm
```

Figure 5.35 shows the factorial process and the stack, step-by-step, as the algorithm runs.

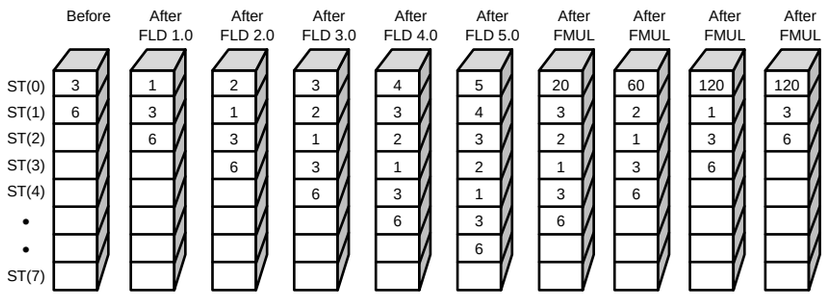


FIGURE 5.35 The stack step-by-step during a factorial calculation.

Playrix Ex. 1023, page 511

08 0672318350 CH05 05•02•2003 2:37 PM Page 487

FDIV Examples

The final class of instruction we're going to look at is the division instruction: FDIV. It's similar to the subtraction instruction in that the order of operands makes a difference; that is, a/b, is different from b/a. As you would expect, the instruction has all the forms: real, integer, and stack popping. Here are the formats for review:

Implied operands are ST(1)/ST(0); also the stack is popped after the division.

```
FDIV
```

The top of stack is implied as the destination and one of the operands.

```
FDIV    op1
        memory32
        memory64
        ST(n)
```

One of the operands must still be the top of stack ST, but the destination need not be.

```
FDIV    op1, op2
        ST, ST(n)
        ST(n), ST
```

The operands op1/op2 *are divided and the result is popped off the top of stack.*

```
FDIVP   op1, op2
        ST, ST(n)
        ST(n), ST
```

The integer referred to by op1 *is converted to a real and divided into the top of stack.*

```
FIDIV   op1
        memory16
        memory32
```

As an example, let's use the formula:

```
velocity = distance/time
```

Let's see how long it takes to get to the moon with your average, run-of-the-mill rocket engine:

```
// Distance to moon: 250,000 miles
float f_dist = 250000.0;
```

Playrix Ex. 1023, page 512

08 0672318350 CH05 05•02•2003 2:37 PM Page 488


```
// Velocity of Value Space Ship: 35,000 mph
float f_vel = 35000;

// the time?
float f_time;

_asm {
   FLD f_dist   // load the distance
   FDIV f_vel    // divide by velocity
   FSTP f_time   // store the result into f_time and pop
   // the stack
   } // end asm

printf("\ntime =%f",f_time);
```

TRICK

Notice how easy it is to mix C/C++ with inline asm and FPU code—pretty cool, huh?

When the code exits, `f_time` will have roughly 7.14 hours in it. Totally unacceptable. I much prefer space/time folding—that is, gravity drive—wherever you want to be, there you are. This was the idea behind the movie *Event Horizon*. Anyway, that's enough for floating-point! We'd better stop before we end up with a book within a book.

Notes on Using the Math Engine

Using the math engine in your programs is just like any other library module. However, there are a couple things to watch out for. First, the math engine is based on some legacy code from `T3DLIB1.CPP¦H`. Thus, you must make sure to include the header `T3DLIB1.H` in your programs that use the math library along with the C++ file `T3DLIB1.CPP`. However, you would do this anyway. Additionally, `T3DLIB1.CPP¦H` is based on DirectX 8.0+, so it needs `DDRAW.H` included for type info. The bottom line is that if you want to make a stand-alone `CONSOLE` math application, you need the following:

Source Files

`T3DLIB1.CPP`

`T3DLIB4.CPP`

Header Files

`T3DLIB1.H`

`T3DLIB4.H`

`DDRAW.H`

Playrix Ex. 1023, page 513

08 0672318350 CH05 05•02•2003 2:37 PM Page 489

Library Files

N/A

Of course, you need to include the normal headers, too, if you want to make a standalone application. Don't forget to `#define INITGUID`; otherwise, the linker won't be able to find the DirectDraw stuff and bark at you even though you aren't doing any DirectDraw stuff with a standalone math application. Remember, `T3DLIB1.CPP¦H` needs DirectDraw types, and `T3DLIB4.CPP¦H` needs `T3DLIB1.CPP¦H`. For example, the test application I used to develop the math engine was a `CONSOLE` application with this header:

```
#define WIN32_LEAN_AND_MEAN

#ifndef INITGUID
#define INITGUID    // you need this or DXGUID.LIB
#endif

#include <windows.h>  // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>
#include <objbase.h>
#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
#include <direct.h>
#include <wchar.h>

#include <ddraw.h>   // needed for defs in T3DLIB1.H
#include "T3DLIB1.H"  // T3DLIB4 is based on some defs in this
#include "T3DLIB4.H"
```

But of course, most of your applications will use the entire game library, so you would include the following:

Playrix Ex. 1023, page 514

08 0672318350 CH05 05•02•2003 2:37 PM Page 490

Source Files

T3DLIB1.CPP	Graphics and utility support
T3DLIB2.CPP	Input support
T3DLIB3.CPP	Sound and music support
T3DLIB4.CPP	Math engine

Header Files

T3DLIB1.H

T3DLIB2.H

T3DLIB3.H

T3DLIB4.H

Library Files

DDRAW.LIB	DirectDraw
DSOUND.LIB	DirectSound
DINPUT.H	DirectInput
DINPUT8.H	DirectInput
DMKSCTRL.H	DirectMusic
DMUSICI.H	DirectMusic
DMUSICC.H	DirectMusic
DMUSICF.H	DirectMusic
WINMM.LIB	Windows multimedia extensions (Visual C++ users only)

I shouldn't have to tell you how to compile, but if you need yet another refresher course, take a look at Appendix B, "Installing DirectX Using the C/C++ Compiler," on compiling DirectX applications.

The Game Console

Because we have a new library module, we should update our game console template to reflect the addition of the new header files (that's about all there is to it). The new game console is named T3DCONSOLE3.CPP and the only thing I did to the file was add this line to the header section:

```
#include "T3DLIB4.h"
```

Other than that, the file is the same as version T3DCONSOLE2.CPP.

Playrix Ex. 1023, page 515

08 0672318350 CH05 05•02•2003 2:37 PM Page 491

Comments on Math Optimization

The code from this chapter is reasonable solid algorithm-wise; that is, we used Taylor series for distance calculations, co-factor expansions to compute determinates, and so forth. The only speed-ups we can get are really in the low-level implementations of the functions by taking advantage of loop unrolling, using algebra to simplify calculations, collect terms, and so forth. Also, you'll notice in many algorithms I first compute the inverse of something and then multiply by it rather then dividing anywhere in the function.

TIP

Division will always suck on computers!

However, as I've mentioned, the best ways to optimize are at the algorithmic level or at the level where you can trade prior knowledge to make assumptions and, hence, gain speed. For example, we're going to be doing a ton of matrix multiplying. It doesn't take too much to realize that we're performing a lot of useless multiplies with most of the transformation matrices since they are so sparse; that is, they have lots of zeros in them. This is a common optimization. But, at some point, we can't make the algorithms any faster and inline assembly is the only way. Nevertheless, I'll be quite surprised if we need to go that far in the book to get the performance we need. I have high hopes for the C/C++ compiler's optimizer!

In closing, in terms of optimization, write clean code and strong algorithms, and then low-level optimize at the end.

Summary

Well, after this chapter, you should have nightmares about vector and matrices forever! That's such a nice thought—in an *Addams Family* kind of way <BG>. In any case, we covered a lot of ground, and you basically got a lot of code dumped on you. Normally, I would feel bad about this much code, but this is a programming book, and everything in this chapter is just the C/C++ version of what we talked about in the previous chapter. The header file macros are the only thing that I didn't cover line by line, but you shouldn't have any trouble figuring them out. Bottom line is, when you're coding using the math engine (if you want to use it), chances are I already wrote the function or macro you're looking for, so make sure to take a look before making it yourself.

The only problem I foresee in the near future is cornering ourselves with any particular system of representation; that is, should we use 3D points or homogenous 4D points? We'll just have to see, but at least we know how to handle them both. In reality, we're going to use 4x3 and 4x4 matrices, so technically we're going to work with homogenous points with an assumed w = 1 or a real w as a 4-tuple. But, I can't decide, so why try?

Playrix Ex. 1023, page 516

08 0672318350 CH05 05•02•2003 2:37 PM Page 492

Finally, I can't overstate how helpful the TI-89/92 calculator was in writing this chapter, because it helped me check my work. If you're going to write any math functions, I highly recommend that you buy a calculator that can perform matrix, vector, and symbolic calculations so you can check your algorithms with test data. When writing the quaternion code, this was even more useful, because quaternions really have no visualization to speak of; that is, they're totally mathematical.

Boy, that was a long summary, huh? Weird…But then again, which way does a water vortex around down a sink if you're at the north pole, south pole, and equator? Think about it….

Playrix Ex. 1023, page 517

CHAPTER **6**

Introduction to 3D Graphics

"You aren't afraid of the dark, are you?"

—Pitch Black

In this chapter, we're going to cover a lot of general ground on 3D graphics, so think of it as an executive summary with details. Then in the remaining chapters, we're going to focus on each of the main areas of expertise relative to 3D graphics. This chapter is an introduction, so I'm not going to get too freaky, but it's also a crash course in 3D, so you can get a feel for the whole process of 3D rather than having to read the whole book before you get the big picture. With that in mind, we're going to cover a number of topics, some with lots of detail, and some with very little detail. By the end of the chapter, you will have a good idea of what the remainder of the book is about. Here's the hit list:

- 3D engine philosophy
- The structure of a 3D game engine
- 3D coordinate systems
- Basic 3D data structures
- Transformations and basic animation
- Introduction to the viewing pipeline
- Types of 3D engines
- 3D tools
- Loading data from the outside world
- Putting it all together

IN THIS CHAPTER

- 3D Engine Philosophy
- The Structure of a 3D Game Engine
- 3D Engine
- Game Engine
- 3D Coordinate Systems
- Basic 3D Data Structures
- 3D Tools
- Loading Data from the Outside World
- Basic Rigid Transformations and Animation
- Review of the Viewing Pipeline
- Types of 3D Engines
- Putting It All Together

09 0672318350 CH06 05•02•2003 2:39 PM Page 494

3D Engine Philosophy

The primary goal of this book is to teach you 3D graphics—not how to use APIs and accelerators. Now, I want to be clear what that means. This book will give you the fundamental knowledge to create your own 3D graphics engine. This means that by the end of the book you will know every single step to creating a 3D engine from data representation, to final rasterization. In essence, if you got a job at NVIDIA, ATI, and so on, you would be able to create an entire engine API and underlying software algorithms that would then be implemented by hardware state machines. On the other hand, this book is *not* going to teach you how to use 3D APIs like Direct3D or OpenGL—they are cool, but if you really want to understand 3D, you need to understand how to write Direct3D or OpenGL yourself—and that's what I'm after. Bottom line is that I want you to be able to sit down on any machine on the planet with nothing more than the address of the video buffer and be able to write a software-based 3D engine. If you can do that, learning any 3D API will be trivial.

The Structure of a 3D Game Engine

There's a lot more to writing a 3D engine than writing a 3D engine! Because we are game programmers, the point of writing a 3D engine is to use it to render a 3D game. What this means is that the game engine, physics system, AI, and most of the game must all be integrated with the 3D engine itself. Otherwise, there's going to be lots of problems. Therefore, when you write a production 3D engine, you need to think about the relationship of the 3D engine and the game engine and how they are going to interact together, or else you will end up with a mess in the end. Figure 6.1 depicts a structural chart of some of the relationships a 3D game needs to have:

- 3D engine
- Game engine
- Input system and networking
- Animation system
- Collision detection and navigation system
- Physics engine
- Artificial intelligence system
- 3D model and imagery database

As you can see, there are a lot of things to think about! In a simple 2D game, collision detection, artificial intelligence (AI), and database issues are all fairly straightforward, but in a 3D game, you really have to think about these things. Of course, in this book we are focusing on the 3D engine part, but the others are obviously of concern. Let's talk about each of these and some issues you have to keep in mind with each.

Playrix Ex. 1023, page 519

09 0672318350 CH06 05•02•2003 2:39 PM Page 495

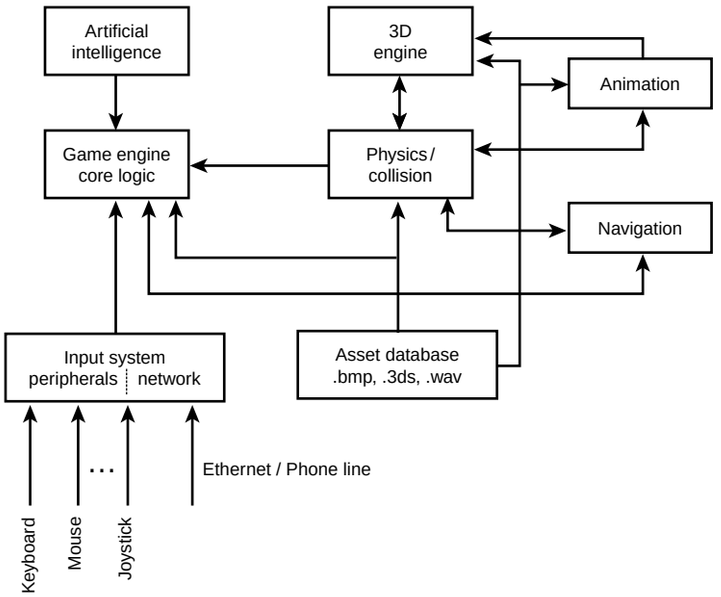


FIGURE 6.1 Structure of a general 3D game engine.

3D Engine

The 3D engine itself is the software that processes the 3D world's data structures, including all the lights, actions, and general state information, and renders the world from the viewpoint of the player or camera. This is relatively straightforward if the engine is designed in a modular way, such that there aren't lots of interrelationships with the 3D engine and the other game subsystems. One of the biggest mistakes you can make when writing a 3D engine is not making it solely responsible for 3D rendering. For example, it's tempting to do the physics modeling in the 3D engine code along with the 3D graphics—this is a big mistake! Another example might be to put some networking hooks in the 3D engine to help simplify your network—that's another no-no.

TIP

The 3D engine does one thing—3D rendering of the data, that's it.

Game Engine

The game engine in a 3D game can refer to a lot of things. Sometimes it doesn't really refer to anything. However, in most cases, the game engine is a system that processes everything, and is the controlling module that sends commands to all the subsystems.

Playrix Ex. 1023, page 520

09 0672318350 CH06 05•02•2003 2:39 PM Page 496

Hence, the game engine needs to be designed so that it's connected to each of the game subsystems. A mistake that is sometimes made is to start on a 3D game with the 3D engine, and then at some point turn a portion of the 3D engine into the main control of the game—that's also a bad idea. From the beginning, design a module that will control everything—call it the "game engine," or whatever. Think of it as the container that holds all the other components and understands every part of the overall system.

Input System and Networking

The I/O and networking systems in a 3D game are notorious for being problems, and they are. The problem is that there is potentially so much data that must be sent over the network for a networked game that making any 3D networked game work is a miracle. Figure 6.2 shows two examples of network gaming models. However, the rule with the network portion of a 3D game is to think about the game first, and keep it separate from the network portion. The networking should not be part of the 3D engine, the AI, or the physics—it should be completely separate. Additionally, the network support for a 3D game needs to be thought out way ahead of any other part of the game. You need to think about what types of data and synchronization techniques you'll use to keep your 3D simulation in step, and then write the entire game with that in mind. Otherwise, you will definitely have a mess in the end.

TIP

If you are writing a 2D game, you can get away with adding networking at the end of development maybe five out of ten times with a lot of ugly code; but in a 3D game, it just won't happen. Decide on network support in the beginning and then create models, simulations, and test cases to test your system before implementing it.

Animation System

This is one of the first systems that may or may not be connected to another in a major way. For example, there are a number of animations that happen in a 3D game:

- Simple motion
- Complex animation
- Physically-based animation

Playrix Ex. 1023, page 521

09 0672318350 CH06 05•02•2003 2:39 PM Page 497

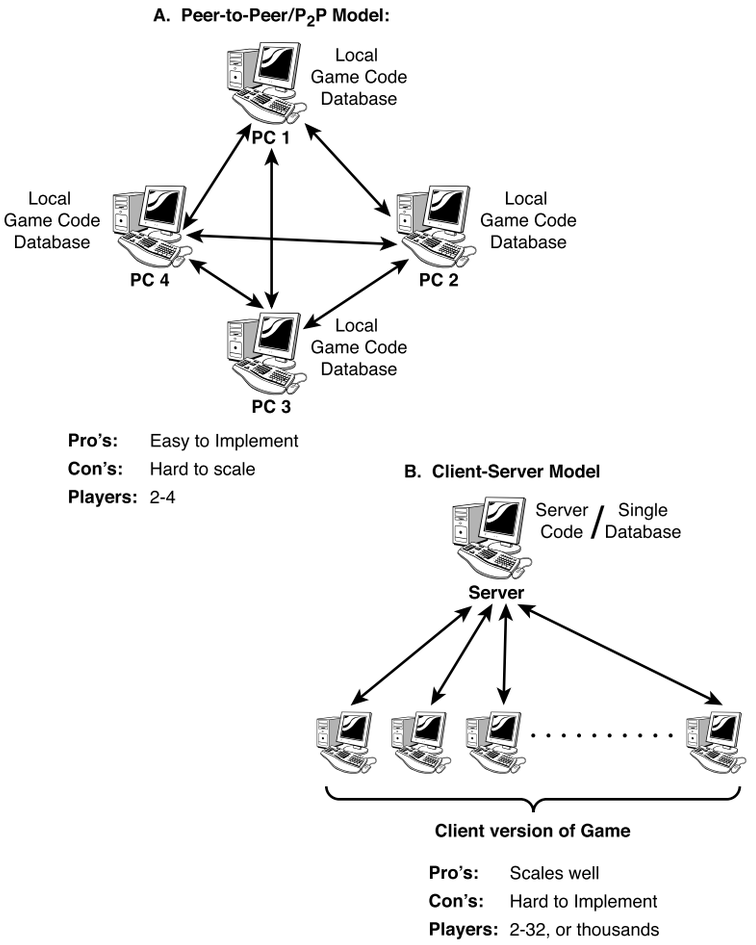


FIGURE 6.2 Network gaming models.

Simple Motion

Simple motion usually consists of the basic translation and rotation of your game objects, as shown in Figure 6.3. This motion can be controlled by data files, AI, patterns, simple logic, state machines, whatever. Nothing magical here. Nonetheless, whatever is driving the motion might be tied to others in a minor or major way. For example, say that you have a game where the artificial intelligence is moving around game units.

Playrix Ex. 1023, page 522

09 0672318350 CH06 05•02•2003 2:39 PM Page 498

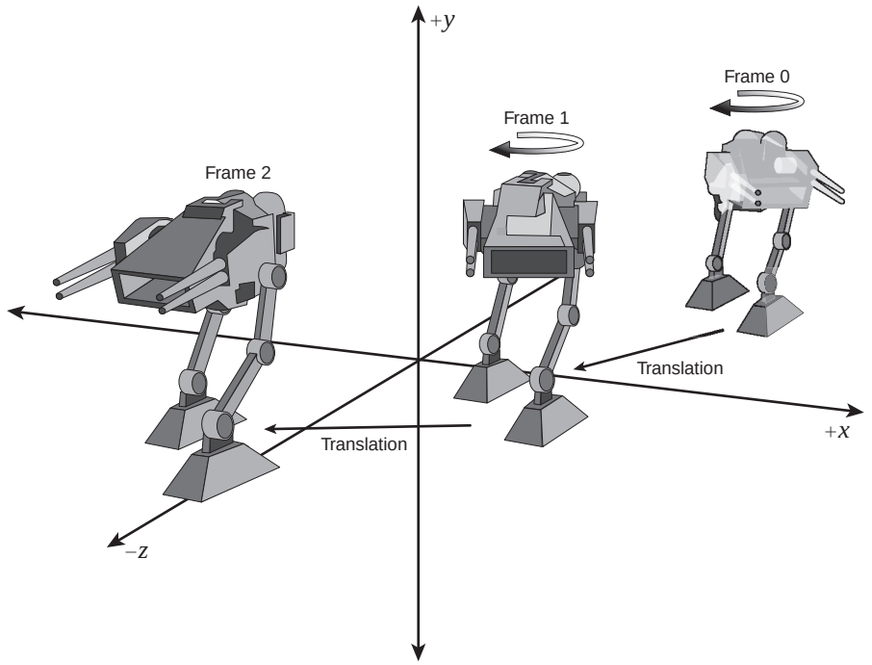


FIGURE 6.3 Simple translation and rotation.

These units move around in the map as shown in Figure 6.4. If the AI has a representation of this map, it can determine the collisions and make sure the units don't go through walls. On the other hand, perhaps the AI could send commands to the physics system, which also handles collision detection, and then it could determine whether the motion is possible. As you can see, in this example at least, you might have the AI, motion, and physics all within one tightly-knit system—and that's okay.

Complex Animation

Complex animation in 3D means a number of things. But, usually it means that you have *articulated-hierarchical* objects, with links that must move in relation to each other. An example seen in Figure 6.5 is a robotic tank with an upper body turret. For simplicity's sake, let's use the tank as the first example. It has a body and a single articulated link—the turret. The turret can rotate around the up axis of the tank body (usually parallel to the ground), but must stay connected to the top of the tank. Now, here's the problem: Is control of the turret going to be completely AI-based, or is it going to be data-based? In other words, suppose we want to make the tank turret do something interesting. It might be very difficult to do that with AI code, but very easy to do it with animation instructions or key framed data. Let's talk about that for a moment.

Playrix Ex. 1023, page 523

09 0672318350 CH06 05•02•2003 2:39 PM Page 499

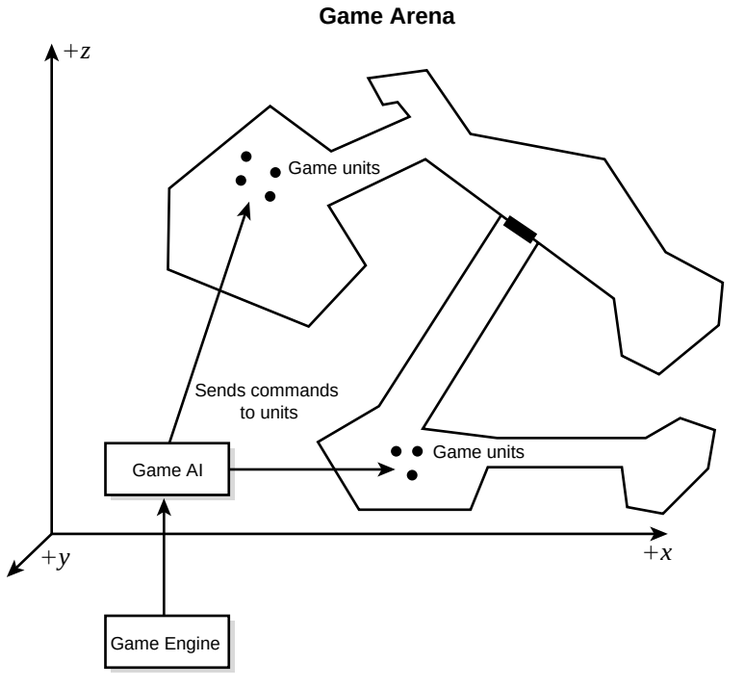


FIGURE 6.4 Artificial intelligence moving game units.

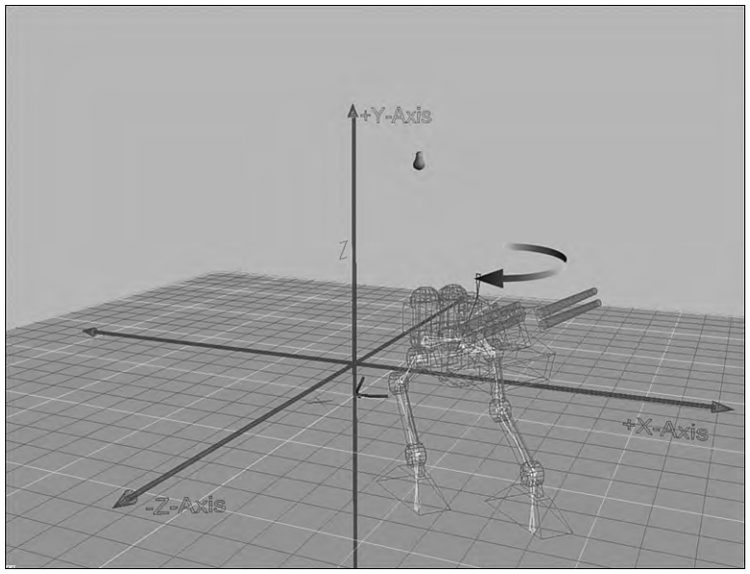


FIGURE 6.5 Animated articulated 3D model.

Playrix Ex. 1023, page 524

09 0672318350 CH06 05•02•2003 2:39 PM Page 500

There are two ways to animate a 3D object that has components or linked objects. The first way is to animate them using a 3D modeling tool, import the meshes, and then animate as a function of time—simply load mesh 1, then mesh 2, then mesh 3, and so on. This technique is similar to standard 2D bitmap animation, except that the data is 3D data representing a 3D model. This method is memory-intensive, but fairly easy to implement. Figure 6.6 shows an example of this. Basically, we model the tank with its turret at one angle, then another, then another, and then save the mesh data output. Then to animate the tank, we load the meshes in and render them one at a time. This is the technique that *Quake* and *Quake II* use.

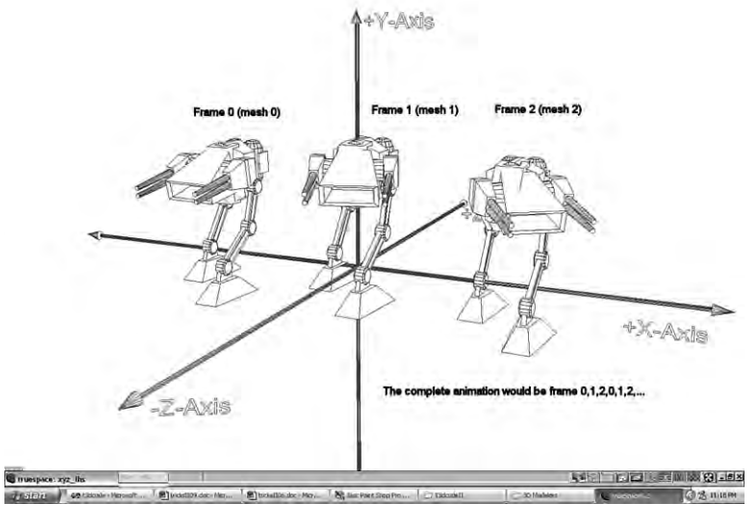


FIGURE 6.6 Using multiple frames to create an animation.

The second way to animate a 3D object is with *motion data*. One popular motion format is the BioVision .BVH format (a little sample is shown in the following code fragment). You can find the specification for this format within the directory for this chapter under `BIOVISION.DOC`. Basically, the data instructs the engine on what links to move relative to others, and the engine then moves each of the subcomponents of the 3D object. This technique is much more flexible, because once you create an animation engine that can read BioVision data, then the data can be anything from a walk to a dance, and your character will do it. This is the technique used in *Quake Arena, Doom III*, *Halo*, and most fighting games, like the *Tekken* series. If you're interested in purchasing motion data, then check BioVision out on the Web at `http://www.biovision.com`.

Playrix Ex. 1023, page 525

09 0672318350 CH06 05•02•2003 2:39 PM Page 501


```
Example Biped-supported BVH File

HIERARCHY
ROOT Hips
{
    OFFSET    20    0.00    0.00
    CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
    JOINT LeftHip
    {
        OFFSET     3.430000     0.000000     0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT LeftKnee
        {
            OFFSET     0.000000    -18.469999     0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            JOINT LeftAnkle
            {
                OFFSET     0.000000    -17.950001     0.000000
                CHANNELS 3 Zrotation Xrotation Yrotation
                End Site
                {
                    OFFSET     0.000000    -3.119996     0.000000
                }
```


The problem with 3D animation is where to draw the line and put in a feedback loop, so that the animations make sense in the physical world? For example, suppose in the tank example that the tank is up against a wall, as shown in Figure 6.7, and a command is executed to rotate the turret around counter-clockwise 90 degrees. Now, the animation was probably initiated by the AI, because the tank wants to shoot an object behind it, but maybe the AI doesn't look ahead that far to see whether the turret will hit something as it rotates. This is a common omission in 3D games—we have all seen objects pass through others. The reason is that the designers didn't really care if once in a while the wheels went under the road, or the arms of the fighters went through the trees, or whatever. However, if we don't want these things to happen, somewhere we'll have to test for it.

The test might happen in the animation system itself—that is, when the turret is instructed to rotate, the animation system checks for possible collisions and then either performs the animation or doesn't. This brings us to yet another problem: Is it the job of the animation system, AI system, or physics system to do this? Again, it really depends on a lot of things. If your game has heavy physics modeling, then yes, the animation system needs to pass this off to the physics system. However, if you have very little physics, maybe it's okay for the animation system to do the testing.

Playrix Ex. 1023, page 526

09 0672318350 CH06 05•02•2003 2:39 PM Page 502

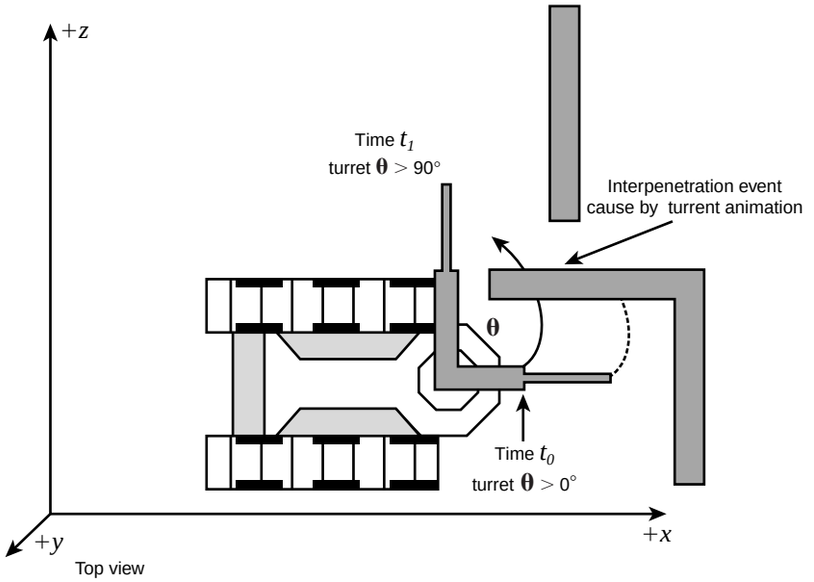


FIGURE 6.7 Physically impossible situations based on nonphysical animation.

I hope that you're seeing that there are a number of complex relationships that occur in a 3D engine!

Physically-Based Animation

This is the most complex of all animations. *Physically-based animation* means that the laws of physics for the most part are controlling how something animates or moves. For example, you might have a 3D fighting game where the fighting moves themselves are pre-canned motion capture data. However, this data is fed into the physically-based animation system, which basically controls physics-based models that perform the animation. They do this not blindly by rotating joints and translating objects, but by directing the model to do these things. Then the physics system determines whether it's possible. This is the best way to perform the most realistic simulation.

Continuing with the fighting game example, let's say that Fighter A throws a punch at Fighter B. The punch is really a set of motion capture data, but instead of just being "played" through the model, the data can be thought of as commands to the model's physics system. So, the physics system starts throwing the punch, then the hand of the punch hits a wall! The physics system then reacts to this by sending a force back through the motion controller and the hand stops dead in its tracks—this is what should happen. As another example, say that Fighter B throws Fighter A. The initial motions of the throw are motion captured, but after Fighter A hits the air, the physics system takes over completely, and he will hit the ground or whatever objects are in the way as the physics unfold based on the environment.

Playrix Ex. 1023, page 527

09 0672318350 CH06 05•02•2003 2:39 PM Page 503

The point is, in a physically-based animation system, the animation and physics are intimately connected. However, once again they have nothing to do with the 3D rendering. By the time the 3D engine renders, all it should know about is a polygon list.

Collision Detection and Navigation System

We have already made reference to the physics system more than once, thereby showing that many systems in a 3D game can be connected. However, sometimes a full physics system is overkill, and all the game really needs is a simpler collision detection and navigation system to run right. We might call this a physics system, but unless it's based on forces, momentum, and other physical quantities, it's really just a glorified collision detection system. As an example, suppose you're creating a *Doom*-style first-person shooter (FPS), and you have almost no physics: The characters are animated by AI along with pre-canned mesh animations, and nothing slides (there's no momentum transfer, and so forth). The only thing you need to take into consideration is object-to-object and object-to-world penetration, as shown in Figure 6.8.

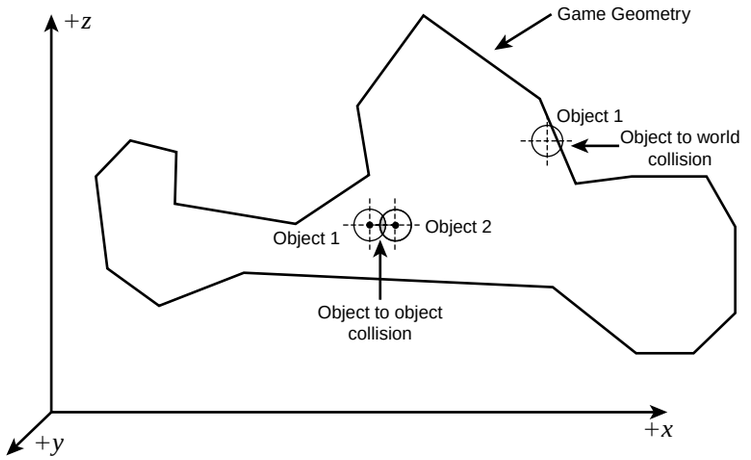


FIGURE 6.8 Object-to-object and object-to-world collision.

Thus, all you need is a simple collision detection system that feeds back the animation system (and/or the AI system) with information about what's happening as the animation system commands the game objects to walk, run, move, or whatever. However, with a system this simple, you might get objects penetrating each other, and creatures sliding on the ground unrealistically, and so forth.

Physics Engine

The physics system in a game can be very simple, as in the collision-only system mentioned previously, or very complex, based on real physics and closed-loop models.

Playrix Ex. 1023, page 528

09 0672318350 CH06 05•02•2003 2:39 PM Page 504

The physics in a game engine really have nothing to do with the visual representation of 3D graphics at all. All the physics engine does is control and respond to actions that the objects in the game make or don't make, and then based on the physics models you've implemented, takes action. For example, a very simple physics model might have the following capabilities:

- Frictional forces and acceleration
- Elastic collision detection and response
- Linear momentum transfer

A good example of this kind of modeling is a simple 3D racing game consisting of a flat track and a number of cars, as shown in Figure 6.9. For each car, we might model the car as a point mass (or maybe a 4-point mass, one point for each wheel). Then as the cars drive around the track, either controlled by the player or the AI, the physics system keeps the cars on the road based on frictional forces only. That is, if a car is moving at 100 mph and then takes a turn at a certain angle, *centripetal force* (center seeking) is computed and compared to the *centrifugal force* (outward), and then the forces are modified appropriately and the acceleration of the car is altered. Furthermore, if a car hits another, they are modeled as spheres, perhaps, a simple elastic collision takes place, and momentum is transferred, but angular momentum is not computed—or it's faked.

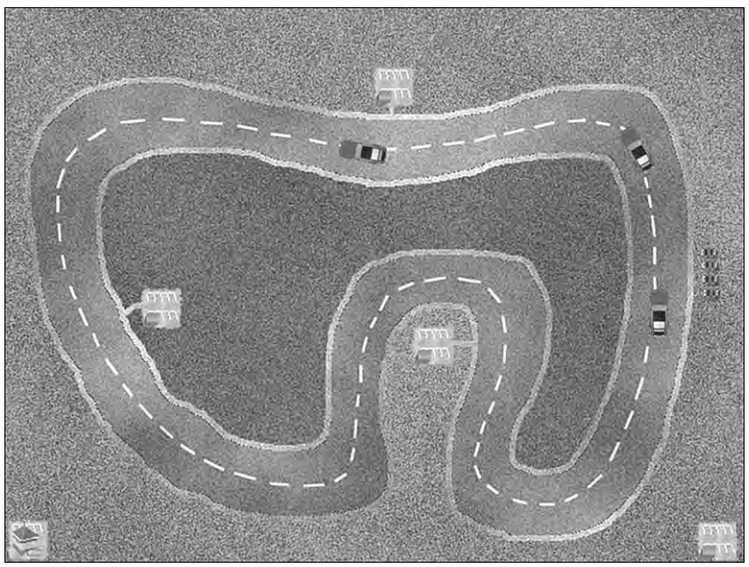


FIGURE 6.9 A simple racing game setup.

Playrix Ex. 1023, page 529

09 0672318350 CH06 05•02•2003 2:39 PM Page 505

The preceding level of physics modeling is usually robust enough for a large class of 2D and 3D games, but if you want reality, you have to go farther. Farther means modeling objects based on their actual geometry, along with using full physics where the physics system is controlling the objects, rather than the AI system controlling the objects and taking feedback from a simplified system of collisions and friction. In a full physics model, each object is continually under simulation, and all the forces on the object are used in the standard physics equations that describe acceleration, velocity, and motion. Along with integration, these equations are used to arrive at the new position of any object, expressed as a function of time based on the forces that are being applied to the object—this is what *dynamics* does for us.

We will cover this in reasonable detail in the physics modeling part of the book. However, keep in mind that a 3D game with this level of simulation must have the animation and AI systems tied into the physics system, so that these other systems respond to the artificial reality imposed by the physics models, and don't continue to send commands to the game units that are impossible to execute because of the constraints of the game's physics.

Artificial Intelligence System

The AI in a 3D game can be an order of magnitude more complex than that of a 2D system or the same level, depending on the modeling of the game units. The problem with AI in a 3D environment is, of course, that it's 3D! Functions like navigation, path finding, and animation all become much more complex. In a 2D game, moving a collection of space ships around isn't that hard, but moving something in 3D is very hard. All the algorithms used in 3D are the same as 2D—deterministic, patterns, planning systems, finite state machines, fuzzy logic, neural nets, and so forth—it's just that a lot more thought has to be put into their implementation.

As an example, take a look at Figure 6.10a and Figure 6.10b. Here we see a 2D environment and a 3D environment from two different shooter games (*Valkyrie* and *Quake III*). The 2D scroller's AI is fairly straightforward because the environment is simple. The objects are nothing more than 2D bitmap animations, most of the object motion is controlled by patterns, tracking logic, and finite state machines. The 3D game works in a similar way, but it's about ten times harder to write! The problem is that everything is in 3D, and even moving the characters is now a large problem. When a 3D character approaches you, it has to look intelligent. This means its motion, animation, and logic have to take the 3D environment into consideration.

These constraints make programming AI for 3D game much harder. Operations like simply moving a unit from one point to another in a 3D world can be overwhelming because of the geometrical makeup of the world. It might be easy for us to "see" a simple path, but for the AI to invent one is not at all easy. Anyway, these are the kinds of issues that are going to come to light in 3D AI programming. Finally, the AI system in a 3D game is usually tied to the animation and the physics system, because the AI must have sensory input from the world, many times in response to its actions.

Playrix Ex. 1023, page 530

09 0672318350 CH06 05•02•2003 2:39 PM Page 506

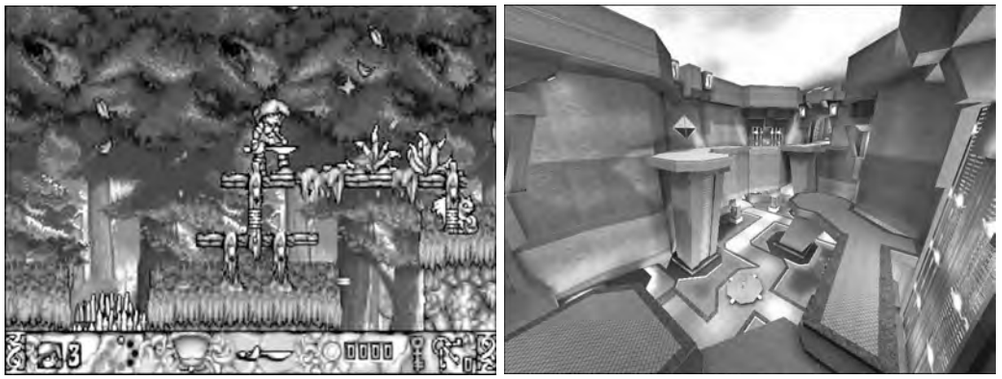


FIGURE 6.10 Examples of 2D and 3D environments.

3D Model and Imagery Database

The last large part of a 3D game is the data. And there's a lot of it! 3D games can have enormous data footprints. This is because of the nature of 3D. For example, a typical 3D game is going to have a number of the following data items:

- 3D meshes for objects
- 2D texture maps and light maps
- 3D worlds
- Motion and animation data
- Game map data

In most cases, the data that take up the most memory are the texture and light maps, because there might be hundreds, if not thousands, of texture maps. And with current texture maps possessing 24-bit color and up to 1024×1024 resolution, that's three megabytes per texture—wow! Of course, many games have user-selectable texture sizes and quality, but even at the 256×256 color level, that's still 65,536 bytes per texture—that's a lot of memory and memory management to go along with it. In this book, we will probably stick to textures in the range of 32×32 to 128×128 at most, just to keep things simple (and fast, because we are using software algorithms), but I think you get an idea of the massive amounts of memory needed for texture storage. Moreover, it's not just memory storage for the textures, but also for the meshes themselves.

Playrix Ex. 1023, page 531

09 0672318350 CH06 05•02•2003 2:39 PM Page 507

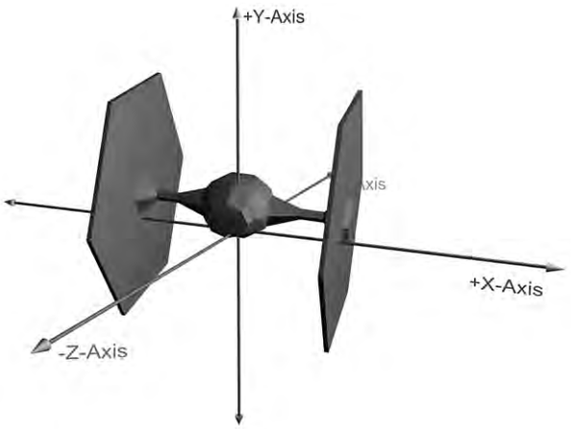


FIGURE 6.11 A low polygon model.

For example, if you decide to go with pre-canned animated meshes rather than motion data controlled animation, the amount of memory can quickly add up to hold an object's mesh animation. For example, suppose you have a character as shown in Figure 6.11 that has around 200–300 polygons and 200 vertices. Let's do the math to see what the storage is on this by making up a simple data structure to hold a 3D vertex that supports both the positional coordinates, along with texture and lighting information:

```
typedef VERTEX3Dptl_TYP
{
float x,y,z; // position
float u,v;   // texture coordinates
float l;     // light value

} VERTEX3Dptl, * VERTEX3Dptl_PTR;
```

That's roughly `6*sizeof(float) = 24` bytes per vertex (on a 32-bit system, at least). Our object has 200 vertices, so that's:

```
24 bytes per vertex * 200 vertices = 480 bytes per mesh.
```

However, we haven't factored in the higher-level stuff like the polygon mesh, color, and so on. Let's just forget about that for now, and continue with the example. Let's say that we want 32 frames of animation for each motion, and we need the following animation meshes:

- Walk
- Run
- Jump

Playrix Ex. 1023, page 532

09 0672318350 CH06 05•02•2003 2:39 PM Page 508

- Duck
- Die
- Shoot

Okay, so that's 6 motions each with 32 frames, for a total of

```
6 motions * 32 frames each = 192 frames of motion.
```

And each frame takes 480 bytes, so we have a grand total of

```
192 frames of animation * 480 bytes per frame = 92,160 bytes.
```

That's a lot of freakin' memory! Multiply that by 32 different creatures in the game, and you have 32*92,160, or roughly three megs of vertices! So, you can see that this stuff adds up. The moral of the story is, in a 3D game, you're definitely going to need efficient data structure, databases, and on-demand loading and unloading of data. Of course, the virtual memory manager of Windows helps a lot with this, but you can't just load everything and then work with it—you will run out of memory.

TIP

You also have to think about bandwidth issues, too. When you are dealing with texture maps and large meshes, just the act of moving things around is going to eat up a lot of memory bandwidth on your system bus. That means that efficient data ordering is important, and that you use something and then you're done with it. You don't want to load object A, use it, discard it and use object B, then load A again. You want to try and order your data so that *cache coherence* is high—meaning you use something as much as possible before loading something else.

In conclusion, a 3D game (or any game, for that matter) is nothing more than a stream of bytes, but the representation of that stream of bytes is really important. Hence, good data structures and a well-thought out plan are always good ideas before you even think about coding. With that in mind, let's get to the 3D part!

3D Coordinate Systems

Now that you've had an introduction to 3D game programming from a high-level point of view, we're going to actually start talking about some 3D stuff, starting with 3D coordinate systems.

Although we've already covered Cartesian (2D/3D), cylindrical (3D), and spherical coordinates (3D), these are more like methods of representing 3D systems themselves. What we're interested in now is how a 3D graphics engine represents actual 3D objects in the various stages of processing.

Playrix Ex. 1023, page 533

09 0672318350 CH06 05•02•2003 2:39 PM Page 509

In general, there can be many different coordinate systems that an object is transformed to as it makes its way to the video screen, through what's referred to as the *3D viewing pipeline*:

- Model coordinates
- World coordinates
- Camera coordinates
- Perspective coordinates
- Screen coordinates

These are the main classes of coordinates, but depending on the 3D engine, you can get pretty crazy with other types of transformations, such as normalizing or warping transformations. Nevertheless, the preceding list is what we'll talk about for now.

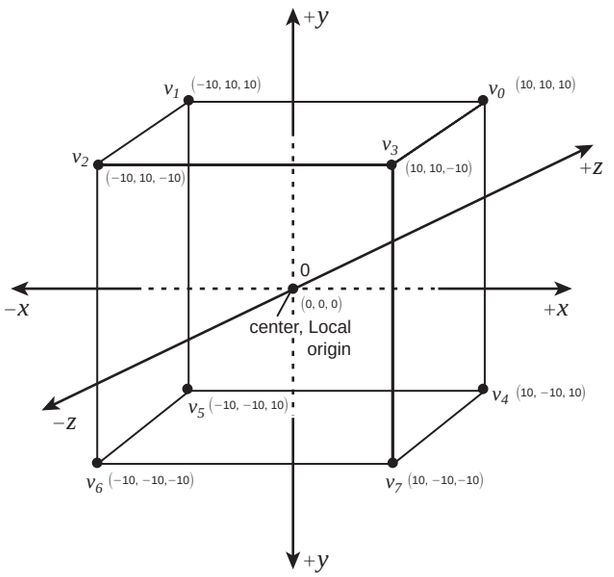


FIGURE 6.12 A basic cube in model/local coordinates for analysis.

Model Coordinates (Local Coordinates)

Model coordinates (also known as *local coordinates*, and I will use them interchangeably) are the coordinates of a 3D entity in its own local coordinate system. That is, every time you create a 3D object for a game, you usually create it with it's own set of local axes, where the center of the object is at (0,0,0) in 3D (or wherever you want the center to be). In

Playrix Ex. 1023, page 534

09 0672318350 CH06 05•02•2003 2:39 PM Page 510

Figure 6.12, we see a cube defined in a left-handed coordinate system. The cube is 10 units on each side, and it has a center (0,0,0). The vertices (points) that make up the cube are

```
Vertex 0: (10,10,10)
Vertex 1: (-10,10,10)
Vertex 2: (-10,10,-10)
Vertex 3: (10,10,-10)
Vertex 4: (10,-10,10)
Vertex 5: (-10,-10,10)
Vertex 6: (-10,-10,-10)
Vertex 7: (10,-10,-10)
```

The idea behind model coordinates is that at some time during the 3D pipeline, you can translate your models to their real positions in the 3D world. This is how a 3D game works. On the other hand, sometimes you might actually want the local origin (0,0,0) of your 3D object not positioned at its geometric center. For example, suppose that you want to create a robot arm model as shown in Figure 6.13. You could define each link as a mesh of polygons, each with its local center at (0,0,0). Whenever the object is translated into world coordinates, it would be centered, so that its center moves to whatever that point is (if I'm losing you, just hold on). However, what if you want the links to rotate or move off center? That is, so that maybe the first link is the shoulder, and the second link is the forearm? There's no rule that states that an object's local coordinates must have (0,0,0) as the center of the object. Therefore, maybe you want the origin (0,0,0) to be at the end of the model's in Figure 6.13. This is actually a more intelligent position to call the origin for these types of objects, since a single point defines the rotation center and the rotating linkage is just a 3D line segment.

Orientation Issues

Lastly, I want to talk about orientation issues with model coordinates. By *orientation* I mean that you must have some kind of convention for defining or loading your 3D models; otherwise, you won't know what side is "up" or "forward." For example, suppose that you define two objects in a 3D game, as shown in Figure 6.14. The space ship has model coordinates such that the front of the ship is pointing in the negative z-axis, and the up direction, or top, is the positive y-axis. The mech, on the other hand, has its front pointing at the positive z-axis. Do you see the problem?

There are two ways to handle this. The first method is to come up with a convention during modeling that states that all models must have their "fronts" pointing in the positive z-axis, and their tops "pointing" in the positive y-axis (or whatever the convention is, as long as you stick to it). The second method is to model your objects anyway you like, but have software compute the major axis of each object (the axis that has the largest length), and align it to the positive z-axis (or whatever axis you choose), along with finding the second major axis and aligning that with the positive y-axis—or whatever, as long as you are consistent. This way, no matter what orientation the object is loaded in

Playrix Ex. 1023, page 535

09 0672318350 CH06 05•02•2003 2:39 PM Page 511

with, you always know that it will be reasonably oriented. My advice is to use the modeling convention rather than letting software do it, because software can only look at the geometry, and doesn't understand what an object really "looks" like.

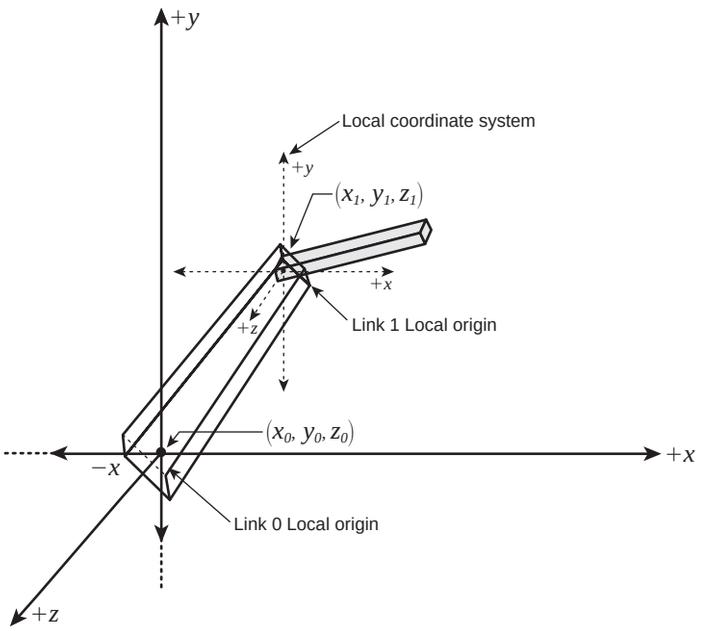


FIGURE 6.13 A robot arm hierarchical linkage.

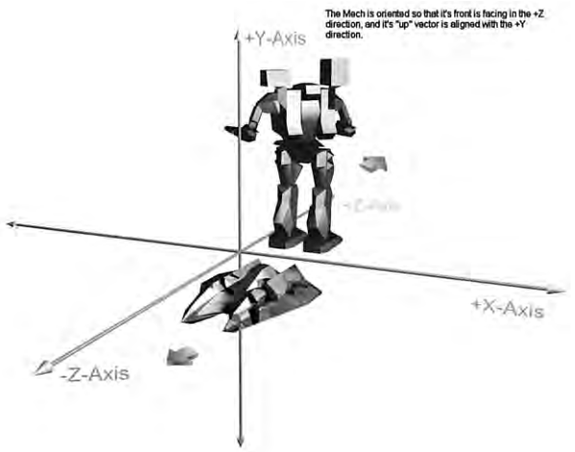


FIGURE 6.14 Arriving at an orientation convention for model loading.

Playrix Ex. 1023, page 536

09 0672318350 CH06 05•02•2003 2:40 PM Page 512

For example, if you load a model of some of the newer 2000+ cars, the fronts look like the back, so you might be driving backwards! Especially that new Thunderbird—yuck!

TIP

Even with an orientation convention during modeling, you might want to rotate, scale, or translate each object during loading. This is a great added feature. For example, you might have a function `Load_3D_Mesh()` that loads an object, but also enables you to transform it during the load. That transformed version becomes the model coordinates that are then passed to the system.

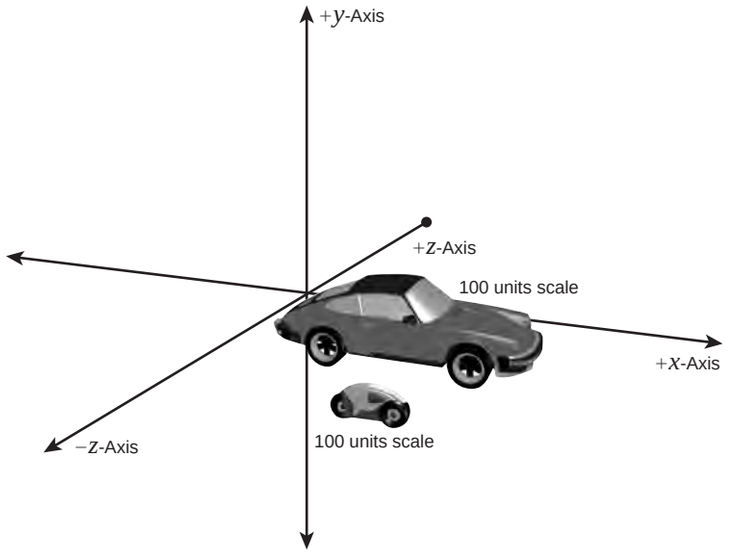


FIGURE 6.15 Differences in scale need to be normalized.

Scale Issues

There's also the issue of the scale of your models. What if one model that represents a car has a radius of 100, whereas another model that represents a light cycle also has the radius of 100, as shown in Figure 6.15? This doesn't make sense, because their 1:1 scales aren't correct. The solution to this problem is to again model all your objects at the same scale, meaning that maybe one unit in the modeling program is always 1 cm in the game engine. Or, you can model everything in a local world of 1×1×1 (normalized coordinates), and then scale the models as you load them. Personally, I just model all my objects using the same scale, so that one unit is something like 1 cm, 1 meter, or 1 km, but I use that same number for all my models as I create them. Again, your loader software function might allow you to specify x-y-z scaling factors, along with x-y-z rotation factors to load

Playrix Ex. 1023, page 537

09 0672318350 CH06 05•02•2003 2:40 PM Page 513

the same model, but change it. This is useful for simple objects, like asteroids. You load in the same model, but scale and rotate it on the fly, so that each loaded mesh seems completely different, which simplifies your 3D modeling work.

A Word on LHS/RHS Coordinate Systems

In a 3D game, you can use either a left-handed (LHS) system or right-handed (RHS) system to define each object with—it's up to you. Of course, you need to be consistent. I prefer to use left-handed coordinates, because all the other transformations are easier (along with projection), and I like the +z axis to go into the screen rather than out. However, sometimes drawing figures on paper is easier with a RHS, because the positive z-axis is pointing toward us. Even if you want to use a RHS for model coordinates and a LHS system for final 3D projection, you can do so with a simple reflection of the z-axis. But why make life hard? I already feel like I'm in the Matrix!

World Coordinates

After you have all your 3D models, you'll want to position them relative to the world. This is where *world coordinates* come into play. They are the actual real coordinates in the virtual universe in which objects move and are transformed. That is, if an object is at (50,100,200), it's *at* (50,100,200)—period! The first step in understanding world coordinates is understanding what their ranges mean in terms of some reference point. That is, how big should the world be? Should it be 1×1×1, 10×10×10, or 1,000,000×1,000,000×1,000,000?

Well, if you're using the convention that one unit in local coordinates is 1 meter, and you want to have a universe that's 100×100×100 km (a good size for a tank or racing sim), that means that your game grid has the following dimensions, given that there are 1,000 meters in a kilometer:

```
Example: World 100 km cubed with 1 meter/unit and origin (0,0,0):

x-axis: -50,000 <= x <= 50,000

y-axis: -50,000 <= y <= 50,000

z-axis: -50,000 <= z <= 50,000
```

That's also a nice number that is easily handled in a float. On the other hand, if you want each unit in model coordinates to represent 1 cm, and have a universe that's 1,000,000 km cubed, you might get into some range issues, because that world would have the following dimensions:

```
Example: World 1,000,000 km cubed with 1 cm/unit:

x-axis: -.5x10^10 <= x <= .5x10^10

y-axis: -.5x10^10 <= y <= .5x10^10

z-axis: -.5x10^10 <= z <= .5x10^10
```

Playrix Ex. 1023, page 538

09 0672318350 CH06 05•02•2003 2:40 PM Page 514

These numbers exceed an integer 32-bit representation, which is roughly +/- 2×10^9 and is getting into the range limits of a standard 32-bit floating number that has any accuracy at all. The bottom line is it's too big. The rule of thumb is that you need to constrain your universe so that you can represent the boundaries of the universe, along with having at least 3-4 decimal places of accuracy.

NOTE

The final scaling and camera transformations come into play along with your models sizes, too, so many times you end up resizing your universe to give it the right "feel" for the game objects, velocities, and scales. For example, I once made a 3D shooter and had a rather small universe (4096×4096×4096), but when the player ran from one end of the universe to the other at 30fps, it took about 15 seconds, which felt about right. A space game, on the other hand, would have to be ten times bigger because the relative velocities of the ships would be much faster. Of course, in reality, they would be thousands of times faster, but in a virtual world, ten times faster and larger would probably suffice.

Placing an Object in World Coordinates

Now that you have a handle on local and world coordinates, let's work through how you would actually transform an object from local coordinates to world coordinates. Remember, the whole point of local or model coordinates is to define the object, but then they must be placed in the world to arrive at the final world coordinates. As an example, take another look at Figure 6.12. Here we see our simple 3D cube object defined in model coordinates—it's a simple cube in this case, with each side having a length of 10 units. Thus, its local coordinates are as follows:

```
Vertex 0: (10,10,10)
Vertex 1: (-10,10,10)
Vertex 2: (-10,10,-10)
Vertex 3: (10,10,-10)
Vertex 4: (10,-10,10)
Vertex 5: (-10,-10,10)
Vertex 6: (-10,-10,-10)
Vertex 7: (10,-10,-10)
```

We could actually define this object using one of our data structures `POINT3D`, as follows:

```
POINT3D cube_model[8] = {
{10,10,10}, // vertex 0
{-10,10,10}, // vertex 1
{-10,10,-10}, // vertex 2
{10,10,-10}, // vertex 3
```

Playrix Ex. 1023, page 539

09 0672318350 CH06 05•02•2003 2:40 PM Page 515


```
{10,-10,10}, // vertex 4
{-10,-10,10}, // vertex 5
{-10,-10,-10}, // vertex 6
{10,-10,-10}, // vertex 7
};
```

And of course, we could define each face that is a polygon that makes up the 3D solid representation as a set of four vertices as follows, if you want to get detailed:

```
Face 0: (0,1,2,3)
Face 1: (4,7,6,5)
Face 2: (0,3,7,4)
Face 3: (2,3,7,6)
Face 4: (1,5,6,2)
Face 5: (0,4,5,1)
```

However, the polygon definitions aren't needed for the transformation from local/model coordinates to world coordinates, so we don't really need the face list.

We have a vertex list of an object in local/model coordinates and we want to transform it into world coordinates. To do this, we simply need to know where in the game world we want the *center* of the object, and then translate the local/model coordinates to this position—simple. Assuming that our world is 1000×1000×1000 and we want to place the object at `(world_x, world_y, world_z)` as shown in Figure 6.16, the following translation algorithm will work nicely:


```
POINT3D cube_world[8]; // used to hold world coordinates

// translate each vertex by (x0, y0, z0)
for (int vertex = 0; vertex<8; vertex++)
    {
    cube_world[vertex].x=cube_model[vertex].x+world_x;
    cube_world[vertex].y=cube_model[vertex].y+world_y;
    cube_world[vertex].z=cube_model[vertex].z+world_z;
    } // end for
```

Notice that we did *not* modify the local/model coordinates of the cube. This is very important. If we did, we would lose our cube model. Rather we transformed them and stored the results in a new vertex list. This might seem like a waste of space, but it can't be avoided. In fact, there might be a number of different versions of coordinates as they are processed through the 3D pipeline. We will try to keep the number of copies to a minimum, but usually there are at least two to three copies of them.

Playrix Ex. 1023, page 540

09 0672318350 CH06 05•02•2003 2:40 PM Page 516

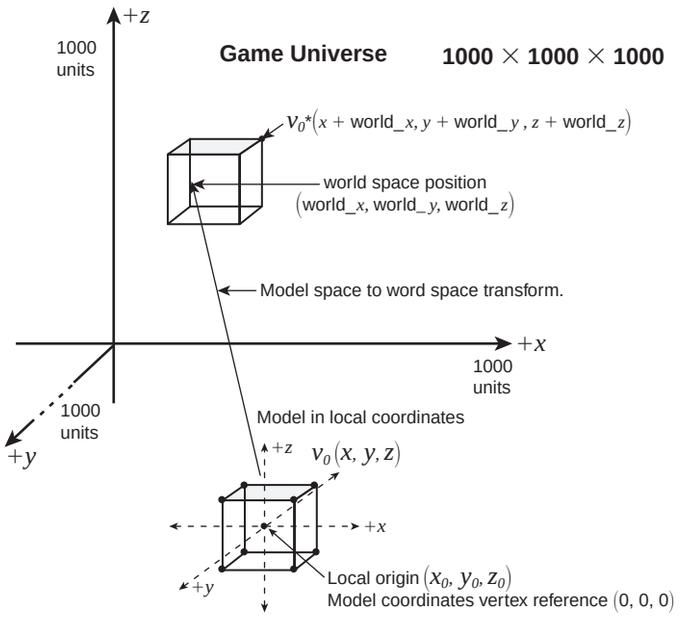


FIGURE 6.16 Placing an object in world space.

In any case, the preceding algorithm is nothing more than a translation. Thus, the local/model to world transformation can be represented with a standard matrix operation; call it T_{mw} for translation model to world. However, there's a gotcha—remember that to support translation in a matrix operation, we need 4D homogenous coordinates, so the matrix would like this:

$$T_{mw} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ world\_x & world\_y & world\_z & 1 \end{bmatrix}$$

NOTE

A word on matrix notation—many times I will name a matrix T with a subscript that further defines the transformation, such as T^{lw}, which might stand for the "local to world transformation matrix." Thus T stands for a general transform. However, other times, if the matrix is a translation or rotation matrix, T means translation and R means rotation. The point is, if you see a matrix named T with a subscript, it's either a general transformation or a translation matrix.

However, if you recall, we have a number of "smart" matrix multiply routines that know how to multiply a 3D vector by a 4×3 matrix, and/or a 4D vector by a 4×3 and 4×4 matrix,

Playrix Ex. 1023, page 541

09 0672318350 CH06 05•02•2003 2:40 PM Page 517

and so on. The point is—to implement a translation via matrix multiplication, you need that fourth row. With that in mind, here's the general matrix operation that must be performed to transform a point $[x_m\ y_m\ z_m\ 1]$ in local/model coordinates to world coordinates.

FORMULA 6.1 Local/Model Coordinates to World Coordinates

$$= [x_m\ y_m\ z_m\ 1] * \mathbf{T}_{mw}$$

$$[x_m\ y_m\ z_m\ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ world\_x & world\_y & world\_z & 1 \end{bmatrix}$$

$$= [x_m + world\_x,\ y_m + world\_y,\ z_m + world\_z,\ 1]$$

$$= [x_w,\ y_w,\ z_w\ 1].$$

Of course, in this example we used 4D homogenous coordinates to make the math simple, but we could have used 3D coordinates as long as we assumed there's a phantom fourth element w equal to 1.0, so that the 4×3 or 4×4 matrix multiply makes some kind of sense. And finally, we would divide (x_w,y_w,z_w) by the last w factor to convert from homogeneous 4D to normal 3D coordinates, but because w = 1.0, there's no point to divide.

Summing up, the transformation from local/model coordinates to world coordinates is rather trivial; you simply translate each object's center to the desired position in the world: (world_x, world_y, world_z). This is in turn accomplished by translating all the vertices in the model and storing the results in a secondary data structure.

As far as the relationship of local/model and world coordinates to the 3D pipeline and the whole scheme of things, you basically transform the object's local coordinates only when you want to scale or rotate the object. Then you place the object in the world with the local/model-to-world coordinate transformation. Figure 6.17 depicts the current 3D pipeline we have up to world coordinates.

Camera Coordinates

Up until now, you've gotten off pretty easy. Now we're going crank up the positron flow… Both local and world coordinates are used to define 3D objects in space. Local/model coordinates define a single entity or collection of entities in their own 3D space and coordinate system relative to (0,0,0). World coordinates define the virtual 3D world in absolute terms—objects are virtually placed in this world. Okay, easy enough, but the point of a 3D game is playing it and moving around a 3D camera! Therefore, we need some way to view 3D objects. This is where *camera coordinates* and the *world-to-camera coordinate transformation* take place.

Playrix Ex. 1023, page 542

09 0672318350 CH06 05•02•2003 2:40 PM Page 518

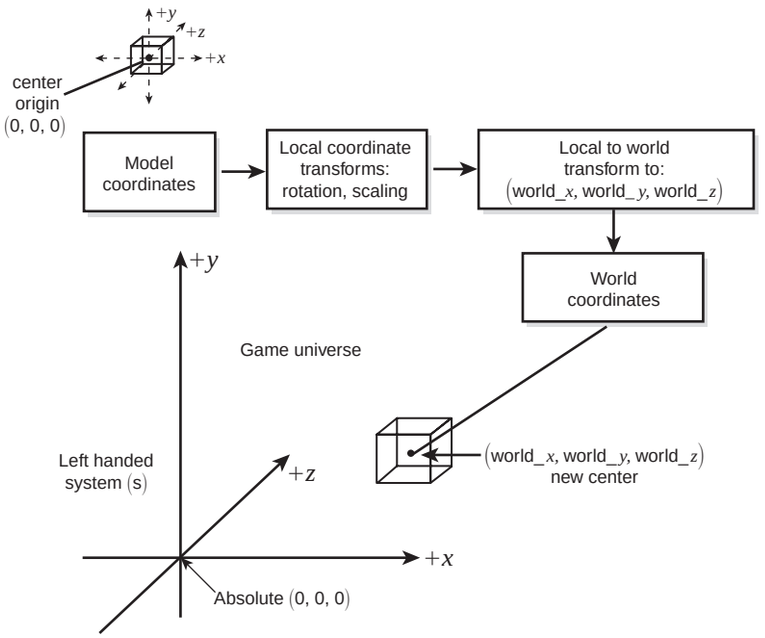


FIGURE 6.17 3D pipeline with local-to-world transform.

The Viewing Frustrum

To view a scene in a 3D world, the most common method is to place a virtual camera at a position in the world and then view the world with a specific field of view. This is shown in Figure 6.18. As you can see, we have a 3D world coordinate system with a virtual camera positioned at (cam_x, cam_y, cam_z) in a direction defined by (ang_x, ang_y, ang_z), or more commonly referred to as *(pitch, yaw, roll)*. Additionally, you see that there's a field of view, or *frustrum*, that the camera can "see in." This can be defined by the *field of view (FOV)* parameters in both the horizontal and vertical directions, which are usually in the range of 60–130 degrees. Additionally, there are near and far clipping planes, which represent the idea that objects beyond these regions are invisible. Only objects within the region are visible. Finally, there's a view plane, which is even more complicated, but ultimately it's the mathematical plane that we are going to project the 3D image on, so we can finally render it on a 2D computer screen. Now, let's talk a little more about the six walls that make up the viewing frustrum before moving on.

Playrix Ex. 1023, page 543

09 0672318350 CH06 05•02•2003 2:40 PM Page 519

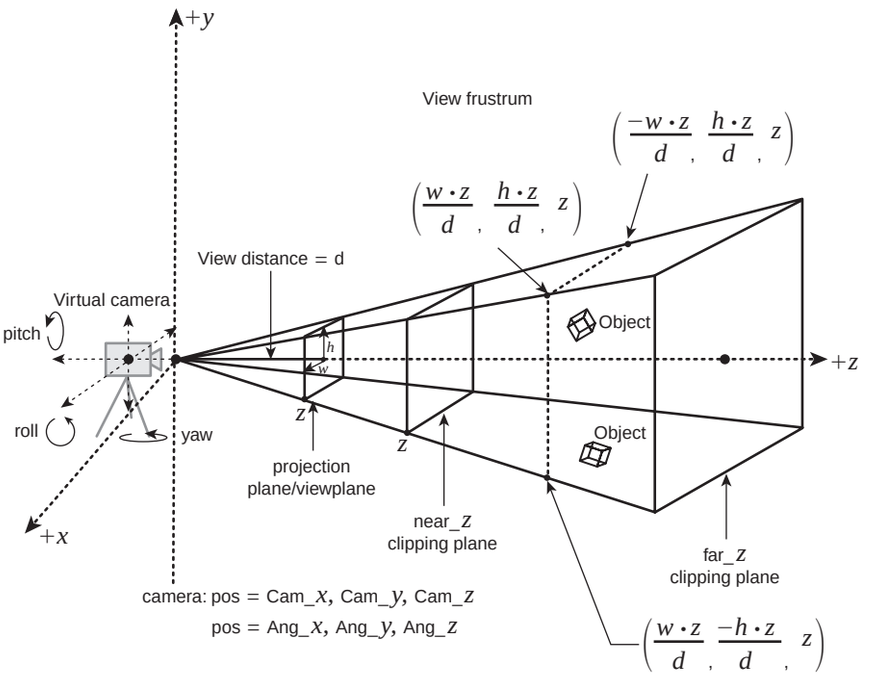


FIGURE 6.18 The 3D viewing frustrum.

Understanding the Near and Far Clipping Planes The near and far clipping planes are located at `near_z`, and `far_z`. Their plane equations are simply `z=near_z` and `z=far_z`, or more generally:

Near z-clip plane: `0*x + 0*y + 1*z = near_z`

Far z-clip plane: `0*x + 0*y + 1*z = far_z`

These planes are perpendicular to the viewing direction and represent the extents that valid 3D geometry can be rendered into. In other words, objects that are located farther than the far clipping plane will not be rendered, and objects that are nearer than the near clipping plane won't be rendered, either. Objects that are farther than the far clipping plane would be so small when rendered and projected that they are pointless to consider, because they would be little dots. Conversely, objects closer than the near clipping plane would be so close to the "eye" of the virtual camera that it would be like mashing your face up against a wall to look at it.

The Viewing Frustrum Walls The viewing frustrum is basically a truncated pyramid defined by the near, far, and side planes. These side planes or walls define the viewing volume of the frustrum. In our case, we are using a 90 degree viewing volume, so each of these planes is at a 45 degree angle to the view direction, as shown in Figure 6.19. Thus,

Playrix Ex. 1023, page 544

09 0672318350 CH06 05•02•2003 2:40 PM Page 520

the plane equations are simply ¦x¦=z, ¦y¦=z. Again, just as the near and far clipping planes, the side walls are used to clip objects, also. Objects that are not within these walls are not visible, and must be clipped and/or removed from the pipeline. (More on this later.)

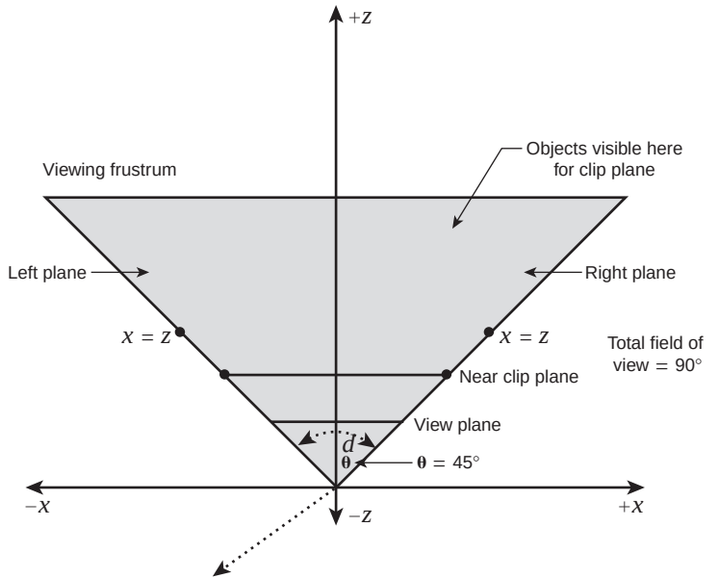


FIGURE 6.19 A simplified top-down view of the viewing frustrum.

The World-to-Camera Relationship The problem is this: We want to transform the objects in the 3D world so that they are relative to the camera. In other words, we want to be able to relate the world coordinates of the objects to the camera in as simple a manner as possible. This is the idea behind the world-to-camera coordinate transformation.

Take a look at Figure 6.19, where we see a camera placed at world coordinate position (0,0,0) looking straight down the positive z-axis with a horizontal and vertical field of view of 90 degrees (it's not necessary to have a FOV of 90, but makes the math easier because the plane equations are trivial). If there's an object in the viewing frustrum, it should be rendered, otherwise it shouldn't. Now, here's the tricky part—99.99999% of the time, the camera is *not* located at the origin looking straight down the positive z-axis (unless of course, you are looking at a DirectX 3D retained mode application of a rotating teapot), the camera is located at some position along with some orientation in the 3D world, usually connected to the front of the player's head.

The question then is this: If we were to artificially move the world so that the camera *was* at (0,0,0), with its orientation looking straight down the positive z-axis, along with the

Playrix Ex. 1023, page 545

09 0672318350 CH06 05•02•2003 2:40 PM Page 521

"up" direction pointing in the positive y-axis direction (that is, view angles equal to (0,0,0)), what transformation would we need to perform on all the world coordinates to make this consistent? This is the magic that the world-to-camera transformation performs and is actually really simple. Well, sorta!

TIP

If you decide to remove the rear end of your car, never under any circumstances take apart the spline gears—you will never get them aligned properly.

The World-to-Camera Transformation

The world-to-camera transformation is a transformation where we move the world around so that from the camera's point of view, the camera is at (0,0,0) looking straight down the positive z-axis, and the objects in the world move appropriately to make this happen. The transformation is accomplished in two steps: *translation*, followed by *rotation*. Let's start with the translation step.

The Translation Step

Take a look at Figure 6.20a. It's a simplified 3D world with a single object in it, along with a camera at some position (cam_x, cam_y, cam_z). We'll assume the camera angles are all (0,0,0) to simplify the example. Now, if we move the camera to (0,0,0) in the world, what would need to happen to the object located at (world_x, world_y, world_z) so that it has the same relationship to the camera? Referring to Figure 6.20b, we see that the camera is now at (0,0,0), but the object needs to be at (world_x, world_y, world_z)—(cam_x, cam_y, cam_z) to keep the relationship between the camera and the object the same. That is, the relative distance and angles have all been preserved. Or in other words, in the frame of the camera, nothing has changed. Only in the frame of the universe has something changed. Killer!

Now with the camera looking down the +z-axis, we can do some math to figure out whether the cube is either contained inside or outside the viewing frustrum, and then some more math to render the cube (with perspective) on the screen if it is. We'll get to that later—the point is, we have solved the translation half of the problem of the world-to-camera transformation.

In other words, the first step of the world-to-camera translation is to translate all world coordinates by an amount equal to the negative values of the camera's position in world space, or (-cam_x, -cam_y, -cam_z). The following code performs this step with our world coordinates from the previous section:

```
POINT3D cube_camera[8]; // used to hold camera coordinates

// translate each vertex by (-cam_x, -cam_y, -cam_z)
for (int vertex = 0; vertex<8; vertex++)
    {
```

Playrix Ex. 1023, page 546

09 0672318350 CH06 05•02•2003 2:40 PM Page 522


```
cube_camera[vertex].x=cube_model[vertex].x-cam_x;
cube_camera[vertex].y=cube_model[vertex].y-cam_y;
cube_camera[vertex].z=cube_model[vertex].z-cam_z;
} // end for
```

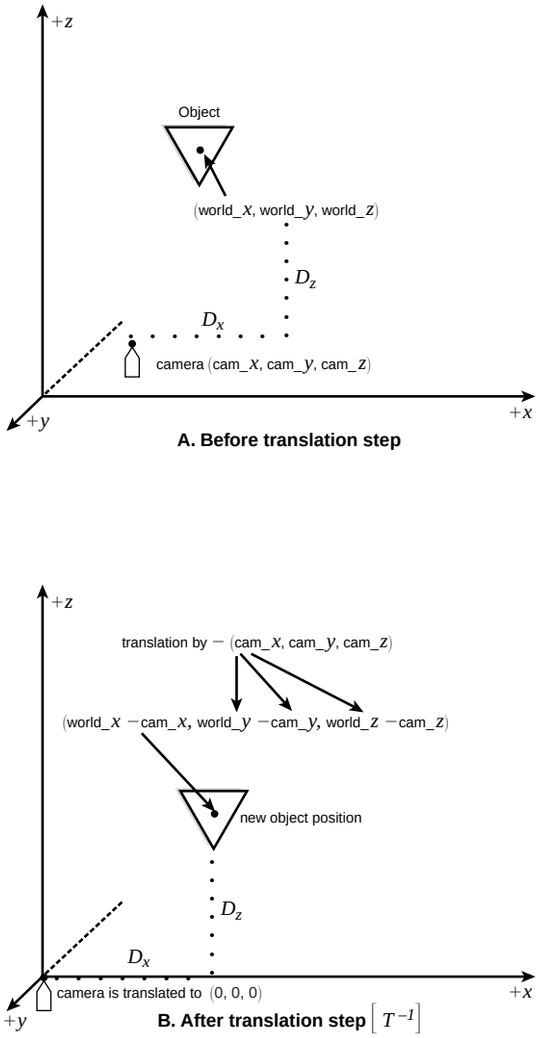


FIGURE 6.20 The translation step of the camera transformation.

Again, we see that yet another set of vertices is needed to hold the camera coordinates. Additionally, the first step of the camera transformation looks just like the local-to-world

Playrix Ex. 1023, page 547

09 0672318350 CH06 05•02•2003 2:40 PM Page 523

transform, so you might think of merging them into one transform. Although we are only half done with the world-to-camera transformation, I will say that it is possible to merge many of the 3D pipeline transformations together via matrix concatenation. However, many times it's better to do things in steps, because we might not transform all objects to camera coordinates. During the 3D pipeline, many objects and/or polygons defined by the vertices might get picked off because of clipping, hidden surface removal, or whatever. However, depending on how the final 3D engine works, it might indeed be true that a single matrix can transform a point from local coordinates all the way to screen coordinates! We'll get to that later, too.

Moving on, we can once again use a matrix transformation to perform the translation step. Assuming that the camera position translation matrix is T_{cam}, the inverse of the matrix T_{cam}^{-1} is formed by inverting each of the translation factors:

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

MATH

Remember, the inverse of any translation matrix is formed by taking the negative of the translation factors in the bottom row of the matrix. Additionally, the inverse of any rotation matrix is formed by supplying the negative angles into each sine or cosine term of the matrix *or* by taking the transpose of the matrix.

So if we were to multiply each vertex in world coordinates against the matrix T_{cam}^{-1}, each vertex would be translated to the proper position to account for the camera being artificially repositioned at (0,0,0). Here's the actual matrix multiply to translate a vertex $[x_w\ y_w\ z_w\ 1]$ from world coordinates to camera coordinates (without taking into consideration the orientation of the camera, of course):

FORMULA 6.2 Translation Step of World-to-Camera Coordinates Transformation

$$= [x_w\ y_w\ z_w\ 1] * \mathbf{T}_{cam}^{-1}$$

$$= [x_w\ y_w\ z_w\ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

$$= [x_w - cam\_x,\ y_w - cam\_y,\ z_w - cam\_z,\ 1]$$

Playrix Ex. 1023, page 548

09 0672318350 CH06 05•02•2003 2:40 PM Page 524

Of course, if you want to code this portion of the world-to-camera transformation with matrices, you could do it with a simple matrix multiply, like this:

```
POINT3D cube_camera[8]; // used to hold camera coordinates

// define translation matrix
MATRIX4X4 Tcam_inv = { {1, 0, 0, 0},
                       {0, 1, 0, 0},
                       {0, 0, 1, 0},
                       {-cam_x, -cam_y, -cam_z, 1} };

// translate each vertex by (-cam_x, -cam_y, -cam_z)
// via a matrix multiply
for (int vertex = 0; vertex<8; vertex++)
    {
    // multiple the vector/point cube_world[]
    // against the translation matrix, note
    // that the matrix multiply assumes that there
    // is a phantom w=1 to make the matrix op
    // valid
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
                         &Tcam_inv,
                         cube_camera[vertex]);

    } // end for
```

CAUTION

Now we have a partial transformation sitting in the `cube_camera[]` vertex array. When you perform matrix multiplies, be sure that the source and destination vertices are different; otherwise, partial values will be used, and the end results will be incorrect. Hence, always make sure that you transform one vertex into another by a matrix—don't transform a vertex by a matrix back into itself!

Now that we have the first part of the camera transformation done, let's work on the second part. Knowing that we need to move the world to the camera, and that the position part was nothing more than performing an inverse or negative translation operation to the world coordinates of each object or vertex an amount (-cam_x, -cam_y, -cam_z), it's fairly reasonable to think that we can do the same thing with the camera orientation.

The Rotation Step

In Figure 6.21a we see almost the same example that we worked with in Figure 6.20. However, in this case the camera is at location (cam_x, cam_y, cam_z), with view angles

Playrix Ex. 1023, page 549

09 0672318350 CH06 05•02•2003 2:40 PM Page 525

(0, ang_y, 0). In other words, the camera has a nonzero heading or *yaw* angle. Thus, it's no longer pointing in the +z-axis direction. If we can figure out how rotating the camera to align it with the +z-axis affects the rest of the world (our little cube, in this case), we have the problem solved. So let's begin …

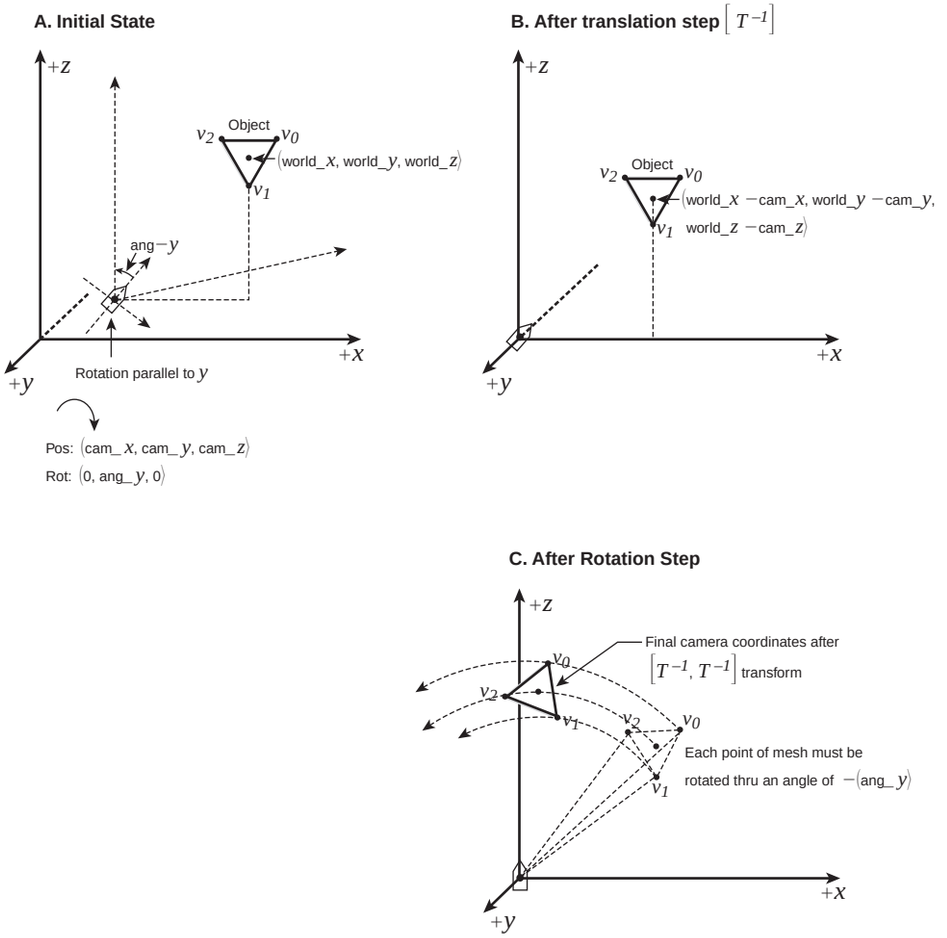


FIGURE 6.21 The rotation step of the camera transformation.

The first thing we do is translate the world coordinates of all objects so that the camera is at located at position (0,0,0) in world space. This is accomplished by translating all objects by the negative position of the camera, or (-cam_x, -cam_y, -cam_z), which is nothing more than multiplying each point in world coordinates against T_{cam}^{-1}.

Playrix Ex. 1023, page 550

09 0672318350 CH06 05•02•2003 2:40 PM Page 526

At this point, we have the situation shown in Figure 6.21b. The camera is located at (0,0,0) and the relative position of the cube is the same, but the camera is still pointing at the location defined by the view angle (0, ang_y, 0). Now all we need to do to align the camera to the +z-axis is rotate the camera about the +y-axis at an angle (0,–ang_y,0), which is the inverse or negative angle that the camera is pointing (0, ang_y, 0). However, if we do this, we need to do the same thing to every vertex in the world, which in this case means the vertices that make up the cube. If we rotate the cube around the +y-axis at an angle (0, -ang_y, 0), the results are shown in Figure 6.21c.

In the figure, we have indeed positioned the camera at (0,0,0) with a view angle of (0,0,0). Nevertheless, the relative position and orientation of the object(s) in the world (the cube) are the same. What *has* changed are the actual absolute vertex positions in world space, but their relationship to the camera and each other has not. This is the complete world-to-camera transformation for a simple viewing system defined by a camera location (cam_x, cam_y, cam_z) and the Euler viewing angles (0, ang_y, 0).

But what about the general viewing angles (ang_x, ang_y, ang_z)? The transformation is the same: You must transform the world coordinates first by the negative or inverse translation matrix T_{cam}^{-1}, followed by a series of rotational transforms that rotate the world coordinates into place by the inverse of the camera orientation. Of course, as there are three angles that make up the camera orientation (ang_x, ang_y, ang_z), there are 3!=3*2*1 = 6 different ways to perform the rotation sequence:

Sequence 1: xyz

Sequence 2: xzy

Sequence 3: yxz

Sequence 4: yzx

Sequence 5: zxy

Sequence 6: zyx

However, most people use either Sequence 3 or 6, because it's more natural to first turn your head (yaw) and then tilt it up and down (pitch). The twist (roll) part isn't very natural (unless you're a dog). However, it's really up to you. The sequence doesn't really mean anything, other than if there were a virtual camera, this is how you would perform the rotation sequence. However, most of the time the camera is never seen in a game—only what it "sees" is viewed; hence, the sequence is arbitrary. In any case, we will assume Sequence 3 for the examples.

With that in mind, we need to perform three rotations: one parallel to the y-axis, which we will call R_{camy}^{-1}; followed by one around the x-axis, which we will call R_{camx}^{-1}; followed by a rotation around the z-axis, which we will call R_{camz}^{-1}. Additionally, the rotations must be the inverse of the camera angles, because we are trying to invert the whole system. Therefore, the entire rotation sequence that must be performed on every vertex to transform it to camera coordinates is as follows:

Playrix Ex. 1023, page 551

09 0672318350 CH06 05•02•2003 2:40 PM Page 527

FORMULA 6.3 The Complete World-to-Camera Transformation for Arbitrary Camera Position (cam_x, cam_y, cam_z) and Orientation (ang_x, ang_y, ang_z)

$$\mathbf{T}_{wc} = \mathbf{T}_{cam}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$$

Of course, you could concatenate all the matrices together into a single matrix and call it T_{wc} for the world-to-camera transformation, and then simply transform each point [x_w y_w z_w 1] by the single matrix T_{wc}, as shown in the following to transform from world-to-camera coordinates:

$$[x_w \; y_w \; z_w \; 1] * \mathbf{T}_{wc}$$

Pretty cool, huh?

TIP

Note that we are using the inverse of the camera translation and rotation matrices. If we wanted to think of the camera as an object, to orient and place it we would transform it into world coordinates with the transformation sequence

$$\mathbf{R}_{camy} * \mathbf{R}_{camx} * \mathbf{R}_{camz} * \mathbf{T}_{cam}$$

Note that the rotation and translation steps have been reversed, and the matrices are not inverted. It's interesting to also note that the camera is really nothing more than a unit vector located at (cam_x, cam_y, cam_z), with orientation (ang_x, ang_y, ang_z).

In practice, you need to compute the matrix $\mathbf{T}_{wc}$, which equals ($\mathbf{T}_{cam}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$). You can do this manually on paper and arrive at a single matrix that you can just initialize, or you can define the independent matrices and then multiply them together one by one and then use the result. Either way is fine, because you will only need to compute the world-to-camera transformation once per frame, and a couple matrix multiplies isn't going to kill you.

However, if you did want to manually compute $\mathbf{T}_{wc}$, it can be computed using algebra to simplify the matrix multiplies, and the resulting matrix will have the inverse translation along with the ($\mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$) rotations all in one matrix. However, I want to hold off on this until later, because it's really nothing more than an exercise in algebra. For now, let's just use our formulas for translation and rotation and let the computer do the work:

```
// given the camera is at (cam_x, cam_y, cam_z) with
// view angle (ang_x, ang_y, ang_z) and a camera
// rotation sequence of y*x*z we have the following:

// here's the translation matrix
MATRIX4X4 Tcam_inv = { {1, 0, 0, 0},
                       {0, 1, 0, 0},
```

Playrix Ex. 1023, page 552

09 0672318350 CH06 05•02•2003 2:40 PM Page 528


```
                        {0, 0, 1, 0},
                        {-cam_x, -cam_y, -cam_z, 1} };

// now we need the rotation matrices for x,y,z
// and remember the angles are ang_x, ang_y, ang_z

// here's the inverse rotation matrix for the x-axis
// remember all you need to do is plug in negative theta
// into the matrix and you will get the inverse rotation
// relative to positive theta
MATRIX4X4 Rcamx_inv = {
        {1,   0,           0,           0},
        {0,   cos(-ang_x), sin(-ang_x), 0},
        {0,  -sin(-ang_x), cos(-ang_x), 0},
        {0,   0,           0,           1}};


// here's the inverse rotation matrix for the y-axis
// remember all you need to do is plug in negative theta
// into the matrix and you will get the inverse rotation
// relative to positive theta
MATRIX4X4 Rcamy_inv = {
          {cos(-ang_y), 0,  -sin(-ang_y), 0},
          {0,           1,   0,           0},
          {sin(-ang_y), 0,   cos(-ang_y), 0},
          {0,           0,   0,           1}};

// finally, here's the inverse rotation matrix for the z-axis
// remember all you need to do is plug in negative theta
// into the matrix and you will get the inverse rotation
// relative to positive theta
MATRIX4X4 Rcamz_inv = {
          {cos(-ang_z), sin(-ang_z),  0, 0},
          {-sin(-ang_z), cos(-ang_z), 0, 0},
          {0,            0,           1, 0}
          {0,            0,           0, 1}};

// now were ready to go, let's multiply them all together
// Tcam_inv * Rcamy_inv * Rcamx_inv * Rcamz_inv

MATRIX4X4 Mtemp1, Mtemp2, Tcam;
```

Playrix Ex. 1023, page 553

09 0672318350 CH06 05•02•2003 2:40 PM Page 529


```
Mat_Mul_4X4(&Tcam_inv, &Rcamy_inv, &Mtemp1);
Mat_Mul_4X4(&Rcamx_inv, &Rcamz_inv, &Mtemp2);

// watch your order!
Mat_Mul_4X4(&Mtemp1, &Mtemp2, &Tcam);

// and now we are finally ready to transform the world
// coords to camera coords

for (int vertex = 0; vertex<8; vertex++)
    {
    // multiple the vector/point cube_world[]
    // against the final camera matrix, note
    // that the matrix multiply assumes that there
    // is a phantom w=1 to make the matrix op
    // valid
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
                         &Tcam,
                         cube_camera[vertex]);

    } // end for
```


MATH

If you want, you can simplify the `sin()` and `cos()` terms in the matrix definitions using the identities `sin(-x) = -sin(x)` and `cos(-x) = cos(x)`. However, I want you to see the negative signs and see how easy it is to compute the inverse of any rotation matrix.

At this point, we can further annotate our 3D viewing pipeline to add the world-to-camera transformation and arrive at the flow diagram shown in Figure 6.22. However, the diagram isn't really accurate. It's missing some intermediate steps, like object removal and clipping, but we will get to them shortly.

Final Word on Camera Coordinates

One last thing I want to bring to your attention is that the world-to-camera transformation depends on the representation of the virtual camera that you select. In this introductory chapter, we are using a fairly primitive camera system that has lots of problems, such as *gimbal lock*.

Playrix Ex. 1023, page 554

09 0672318350 CH06 05•02•2003 2:40 PM Page 530

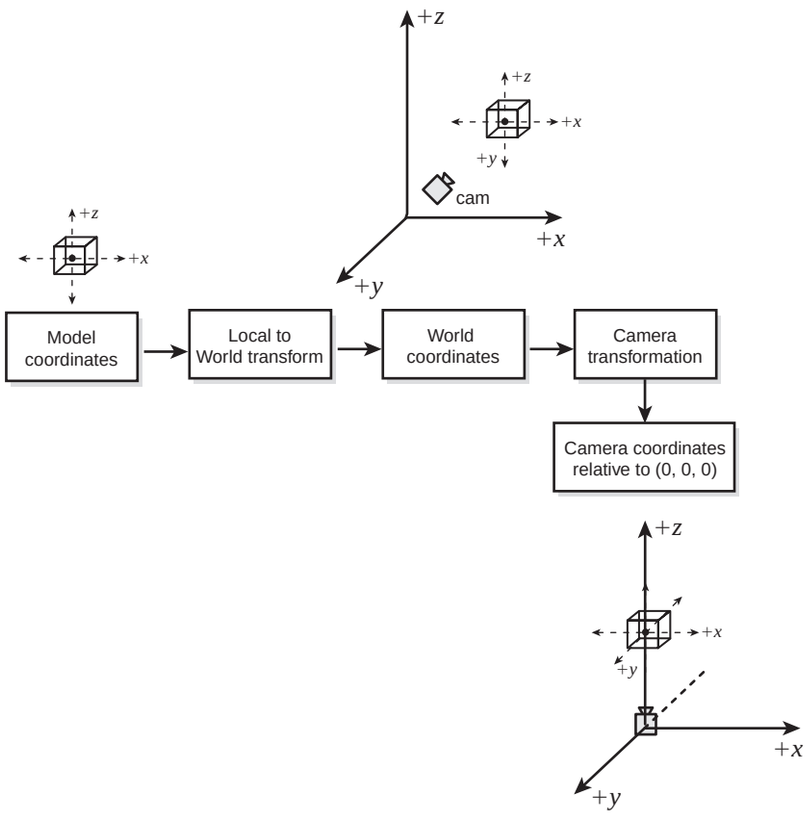


FIGURE 6.22 The 3D pipeline with the camera transformation added.

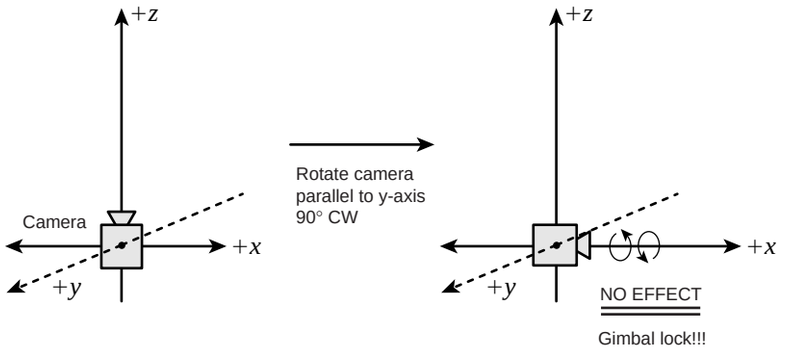


FIGURE 6.23 Gimbal lock.

Playrix Ex. 1023, page 555

09 0672318350 CH06 05•02•2003 2:40 PM Page 531

Gimbal lock occurs when you rotate one axis to the point that it becomes aligned with another, and further rotation on that axis results in no motion at all. For example, take a look at Figure 6.23. In the figure, if we first rotate the camera around the y-axis 90 degrees so that it aligns with the x-axis, trying to rotate around the x-axis does nothing—this is gimbal lock.

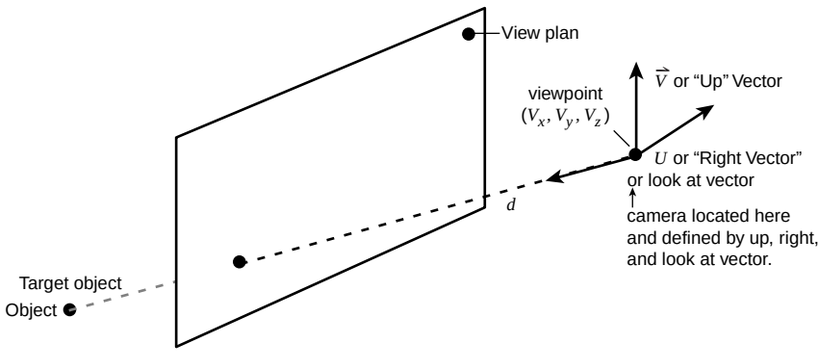


FIGURE 6.24 The UVN camera system.

Alas, there are better camera systems, based on a more natural representation of a camera, taking into account its position and orientation in terms of a focal point (or "look at" point, as shown in Figure 6.24), but we will get into that in more detail in the next chapter.

Hidden Object/Surface Removal and Clipping

Everything you have just learned is correct, but there are two topics I purposely left out because they really don't have anything to do with coordinates: *hidden surface/object removal* and *clipping*. Both of these topics are going to take some time to explain, so right now all I want to do is make you aware that we need to consider them in the 3D pipeline. Therefore, let's take a minute to talk about them, beginning with hidden object/surface removal.

When you perform the world-to-camera transformation (or before you perform it), there are a number of tests you can perform to determine what objects are visible to the camera. These tests are generally classed as hidden surface removal. That is, we need to determine what objects and/or geometry are visible to the camera, so that we don't mistakenly try to render them (or more importantly, not to push the objects through the camera transformation in the first place). With that in mind, two tests are generally performed: back-face removal and the bounding spheres test.

Playrix Ex. 1023, page 556

09 0672318350 CH06 05•02•2003 2:40 PM Page 532

Back-Face Removal

In Figure 6.25, we see an object and we see the camera. However, right now the coordinate system is still in world coordinates. The back-face removal test is performed before the world-to-camera transformation in world-space. This test is used to limit the number of polygons that are transformed from world-to-camera coordinates that would be impossible to see because they are facing away from the viewpoint.

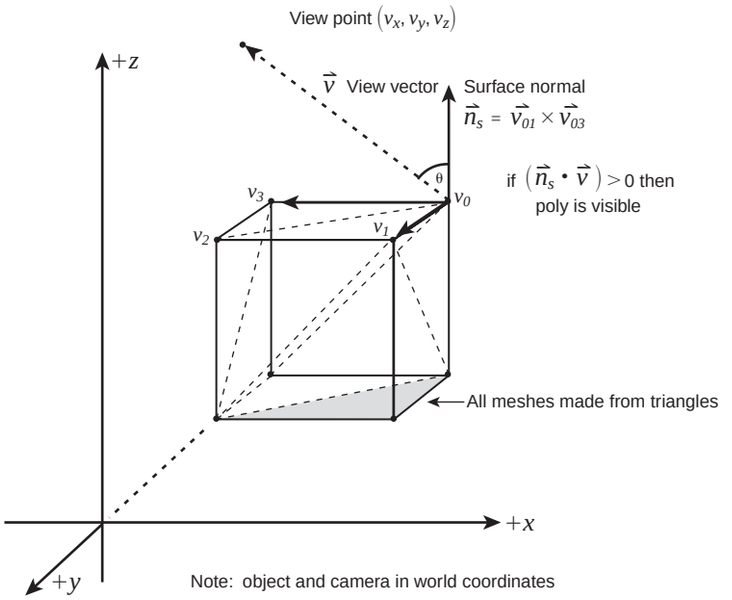


FIGURE 6.25 Back-face removal setup.

The logic of the back-face test works in the following way: All polygons that make up each object are labeled in a consistent way, either clockwise or counterclockwise (it doesn't matter). Then a surface normal $\mathbf{n}_s$ is computed with that knowledge for each polygon, and this normal is tested against the viewing vector **v**, as shown in Figure 6.25. If the angle between the normal and the viewing vector is less than or equal to 90 degrees, the polygon is visible to the viewer (of course, for two-sided polygons, this test doesn't work). Using this test, we can trivially reject half the geometry of each object.

The test itself is performed with the dot product, based on the fact that any pair of vectors that are dotted together have the following properties:

- u.v = 0, if and only if the angle between u and v is 90 degrees or pi/2 radians.
- u.v > 0, if and only if the angle between u and v is less than 90 degrees or pi/2 radians.
- u.v < 0, if and only if the angle between u and v is greater than 90 degrees or pi/2 radians.

Playrix Ex. 1023, page 557

09 0672318350 CH06 05•02•2003 2:40 PM Page 533

Thus in our data structures that define objects as meshes of polygons (which we will get to shortly), we have a flag that indicates that a polygon is one-sided, so we can apply this test and filter out a great deal of the polygons from the expensive world-to-camera transformation with the following pseudo-code:

```
if (n_s . v) > 0)
   {
   // poly is visible
   } // end if
```

NOTE

You might decide that polygons that are exactly 90 degrees relative to the view direction are visible, too—that is, polygons that are on edge, or *scathing*. However, this can cause problems with rendering, because these polygons will usually have a width of one pixel.

Bounding Spheres Test

After you perform the back-face removal step and filter out as many polygons in world coordinates from the camera position as you can, you then transform all the objects from world coordinates to camera coordinates. However, you can do one more test before you perform the back-face removal step at an even higher level. In fact, you should probably do this test before you perform the back-face removal, so you can throw entire objects away. However, this test only works if you have objects that are partitioned in convex areas. In other words, if you have a gameworld that takes place in space, for example, you can locate each asteroid and ship and put bounding boxes (or spheres) around them. On the other hand, with an indoor game composed of a single mesh, the world won't work with this type of test. However, you can surely partition rooms or areas and so forth. The point is simply this: Try to prevent large areas of the game geometry from being processed if they aren't visible whenever possible.

The idea of the test is to create a bounding sphere around all the objects in world space, as shown in Figure 6.26. Then you perform the world-to-camera transformation with only the center of the spheres (a single point). Then you test to see whether the spheres are contained within the viewing frustrum. If they aren't, you can throw the entire object away. If the spheres are partially contained in the frustrum, the test is inconclusive, and further testing is needed.

Therefore, if you have a world that is composed of objects composed of polygons, your first step might be to compute a bounding sphere around each object in world-space, performing the world-to-camera transformation only on the spheres (which are defined by a single point), and then testing whether the spheres are within the camera's viewing frustrum. Objects that aren't in the viewing frustrum at all would be thrown away. You must continue on with the back-face removal and finally perform the world-to-camera transform for the remaining polygon mesh geometry that is neither a back-face nor out of the viewing frustrum.

Playrix Ex. 1023, page 558

09 0672318350 CH06 05•02•2003 2:40 PM Page 534

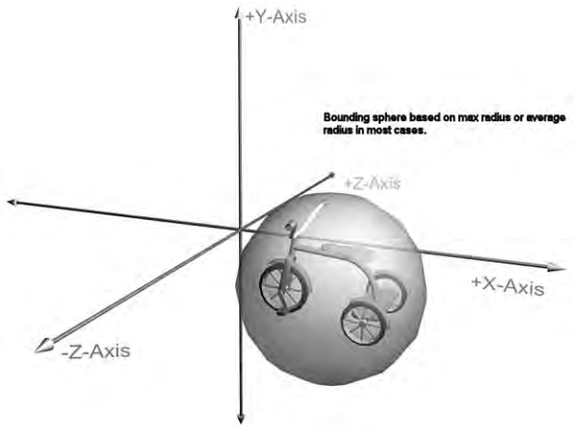


FIGURE 6.26 Placing a bounding sphere around objects to test for view frustrum culling.

Although I don't want to get into the details of the bounding sphere algorithm until we have a working wire-frame 3D engine, let's take a quick look at the math. The problem is very simple. Assume that we have an object **O**, which is a collection of vertices. We compute a bounding sphere around the object by determining the farthest vertex from the center of the object (this can be done ahead of time in a loading phase, rather than in real time), which is in essence its maximum radius. A possible algorithm is shown in the following, which you would run in local/model coordinates in most cases:

```
POINT3D cube_model[NUM_VERTICES]; // our 3D model

float max_radius = 0; // set to 0
float curr_radius;

// iterate thru the vertex list and keep updating
// the radius with the largest radius
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
    {
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;

    // have we found a larger bounding sphere?
    if (curr_radius = sqrt(x*x + y*y + z*z) > max_radius)
        max_radius = curr_radius;

    } // end for
```

Playrix Ex. 1023, page 559

09 0672318350 CH06 05•02•2003 2:40 PM Page 535

Of course, I'm being pretty brutal by using `sqrt()`, and I could get along with an approximation, or even by using the 1-norm. That is, simply find the vertex that has the largest distance^2, and then take the `sqrt()` to find the real distance:

```
POINT3D cube_model[NUM_VERTICES]; // our 3D model

float max_radius = 0; // set to 0
float curr_radius;

// iterate thru the vertex list and keep updating
// the radius with the largest radius
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
    {
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;

    // have we found a larger bounding sphere?
    if (curr_radius = (x*x + y*y + z*z) > max_radius)
        max_radius = curr_radius;

    } // end for

// compute real distance now
max_distance = sqrt(max_distance);
```

TIP

The preceding trick works because if |a| > |b|, then $a^2 > b^2$, and the similar argument |a| < |b|, then $a^2 < b^2$.

Either way, it doesn't really matter, because you would run this algorithm in an initialization phase. After we compute `max_radius`, then we can do some work to create the bounding sphere: With the center of the object in world coordinates, calling it the point **p0**(x0, y0, z0) and its radius `max_radius`, we perform the world-to-camera transformation on the point **p0** as follows:

$$\mathbf{p1} = \mathbf{p0} * \mathbf{T}_{wc}$$

Now, **p1** is the center of our object's bounding sphere in camera coordinates, as shown in Figure 6.27. Here's the fun part: We define six other points relative to **p1** parallel to the +-x, +-y, and +-z axes that are a distance `max_radius` from **p1**, which define the boundaries of the bounding sphere S around **p1**. Then we are ready to determine whether the bounding sphere is within the viewing frustrum with some simple tests.

Playrix Ex. 1023, page 560

09 0672318350 CH06 05•02•2003 2:40 PM Page 536

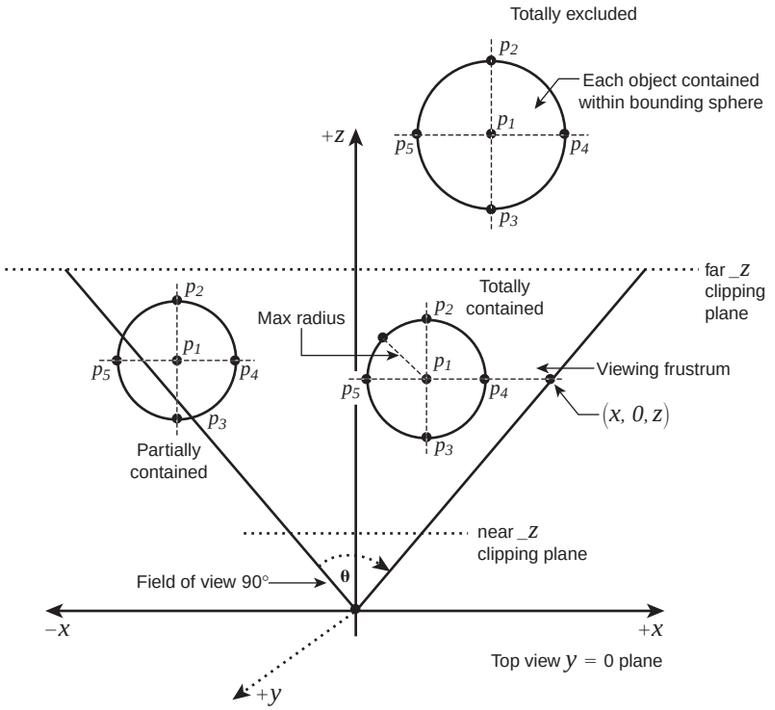


FIGURE 6.27 The details of the bounding sphere tests.

In Figure 6.27 we see a top view of the x-z plane. The tests for the y-z plane are identical, so we need not belabor them. There are a number of cases we need to consider—we need to compare the bounding sphere to the viewing frustrum and determine whether it's totally outside, totally inside, or partially inside. If the bounding sphere S is partially inside or totally inside, we simply include the entire sphere's mesh in the data stream and then perform further tests. At this point, you can either try and determine whether the bounding sphere is inside or partially inside the viewing frustrum or completely outside—it's up to you. However, sometimes using negative logic is cheaper, so that's the approach I will take in this example.

Okay, the problem can be stated as follows in 2D: We have a circle that is defined by a center $\mathbf{p}_1$ (x1,z1), along with four points that define its boundaries in the x and z directions, which are nothing more than

```
Center: p1(x1,z1)

Exterior points:
```

Playrix Ex. 1023, page 561

09 0672318350 CH06 05•02•2003 2:40 PM Page 537


```
p2(x1, z1+max_radius)
p3(x1, z1-max_radius)
p4(x1+max_radius, z1)
p5(x1-max_radius, z1)
```

Theorem: If any of the points **p1**–**p5** are in the viewing frustrum, then the bounding circle/sphere must be in the viewing frustrum.

Corollary: If none of the points **p1**–**p5** are in the viewing frustrum, then no portion of the bounding circle/sphere can be in the viewing frustrum.

Take your pick, but I usually use the corollary and try to trivially reject the entire sphere. With that in mind, we need to really look at the viewing frustrum and come up with some equations so we can quickly plug in the vertex values **p1**-**p5** and make decisions. Referring to Figure 6.26, the near and far clipping planes are fairly easy to test against. If either of the following conditions are true, the object can't possibly be in the viewing frustrum:

Far clipping plane conditional:

```
If (p3.z > far_z)
   {
   // object is not in viewing frustrum
   } // end if
```

Near clipping plane conditional:

```
If (p2.z < near_z)
   {
   // object is not in viewing frustrum
   } // end if
```

We always hope for those! The problem is that the sides of the viewing frustrum (right, left, top, bottom) are defined by general planes. But again, we can use similar triangles or the plane equations along with a dot product to determine the relative position of the bounding sphere to the planes. For now, let's just stick to the 2D case and assume a 90 degree viewing volume. With those assumptions, the problem boils down to determining whether a point **p** (x,z) (for the x-z top-down case) is within the triangle defined by the left and right sides of the viewing frustrum. This can be solved with similar triangles.

However, if you're paying attention, there's an easier way. Remember that for a viewing volume of 90 degrees, the plane equations for the right and left sides of the frustrum are

$$|x| = z$$

Therefore, for a point **p** (x,z) that we want to test against the right clipping plane, if $x < z$, it means that the point **p** is on the left half space of the plane. If $x > z$, it means that the point is on the right half space of the plane, and $x = z$ means the point is right on the

Playrix Ex. 1023, page 562

09 0672318350 CH06 05•02•2003 2:40 PM Page 538

clipping boundary itself. A similar argument can be used for the left clipping plane, but the sign of x is simply inverted. With that in mind, we can complete our rules for the left and right clipping planes for 90 degree FOV:

Right clipping plane conditional:

```
if (p5.x > p5.z)
   {
   // object is not in viewing frustrum
   } // end if
```

Left clipping plane conditional (note the sign and comparison inversion because the left clipping plane has the equation –x = z):

```
if (-p4.x > p4.z)
   {
   // object is not in viewing frustrum
   } // end if
```

The same argument can be made for the top and bottom clipping planes; thus the general rules to test whether a point **p** (x,y,z) is outside the viewing frustrum are as follows:

```
if ((z > far_z)   ¦¦ (z < near_z) ¦¦ // z-axis
    (fabs(x) < z) ¦¦ // x-z plane
    (fabs(y) < z))   // y-z plane
    {
    // point not in viewing frustrum
    } // end if
```

Now all you have to do is pick the right points to test—in this case, **p1**–**p5**—with the preceding rule.

NOTE

All this stuff sounds good, but remember one thing: So much depends on your data structures and representations of your objects. It might be that your whole game is composed of free polygons, and there's no real containable "objects." Perhaps it's pointless to remove back-faces, because meshes are based on vertex lists that must be transformed anyway, and there's no real "polygon" list that is a separate entity from the vertices that make up the object that the polygons skin. Or maybe all your objects are semitransparent, and thus all polygons have two sides. Alas, in many cases backface removal may not make sense.

Limitations of Object Removal

Before moving on, I want to bring up one point about object removal, which is this rule: If an object's bounding sphere is not in the viewing frustrum, the object itself cannot

Playrix Ex. 1023, page 563

09 0672318350 CH06 05•02•2003 2:40 PM Page 539

possibly be in the viewing frustrum. However, the opposite is not true: If a portion of the bounding sphere is inside the viewing frustrum, that does not necessarily mean the object is! Why? Because the bounding sphere might not be a good representation of the object. For example, take a look at Figure 6.28, where we see a very long object's bounding sphere and the viewing frustrum. Interestingly enough, the bounding sphere is in the viewing frustrum, although the object is not! This doesn't mean that the bounding sphere test doesn't work; it just means that it might not filter out all objects that are not in the viewing frustrum. Hence, sometimes bounding geometries that are geometrically more appropriate for the object are used, such as cubes or parallelepipeds.

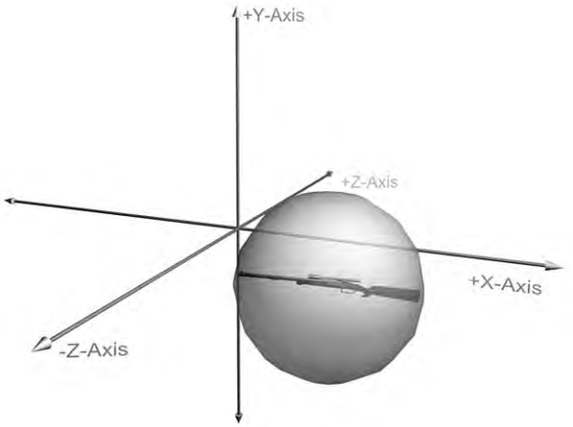


FIGURE 6.28 A bounding sphere might not be the best choice.

That's about it for the quick tour of object and back-face removal. Now take a look at Figure 6.29 to see our updated 3D pipeline—notice we're still a long way from the screen!

Perspective Coordinates

Now we're getting somewhere! Let's reiterate the major stages of the 3D pipeline:

Stage I: All 3D objects are usually defined in their own local space with their own local coordinate system. This representation is called the local or model coordinates representation.

Stage II: After you have all the 3D models that you want to place in the virtual 3D world, you place them by transforming the model coordinates to world coordinates. This is achieved with the local-to-world transformation. Furthermore, this transformation is nothing more than a translation of the vertices that make up each model to the desired center position at which you want the object placed in the world. Be careful not to frag your model coordinates in the process—you need another array to hold the results!

Playrix Ex. 1023, page 564

09 0672318350 CH06 05•02•2003 2:40 PM Page 540

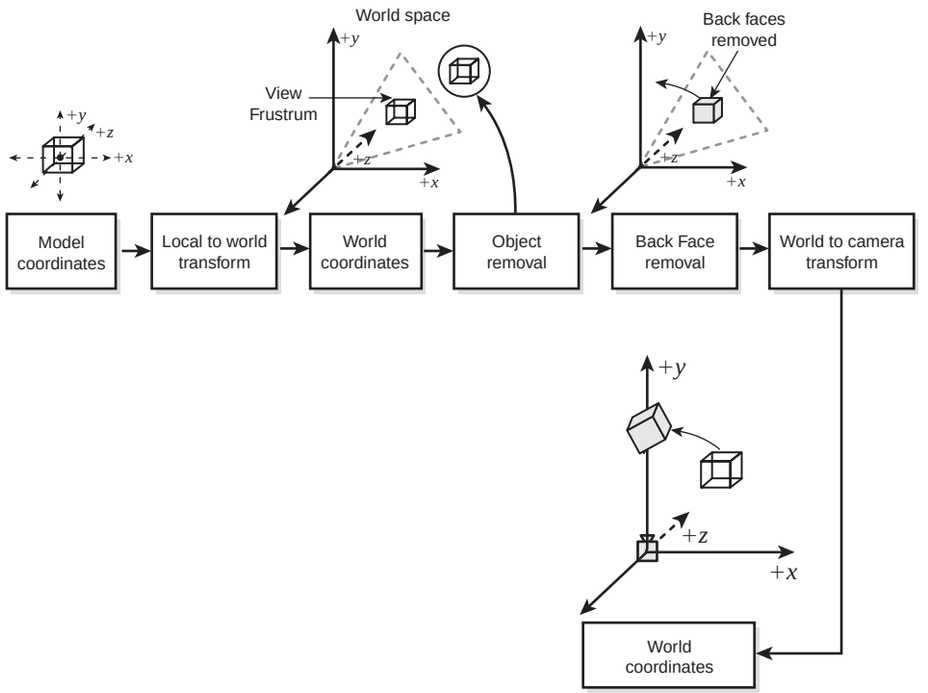


FIGURE 6.29 The 3D pipeline with back-face and object removal added.

Stage III: After you have all your objects placed in the world and they are defined with absolute world coordinates, you need to view the world through a virtual camera. There are a number of ways to represent the camera, but a position and orientation will suffice.

Stage IV: You might have a hidden object/surface back-face removal step at this point that minimizes the amount of geometry that is transformed to camera coordinates. You might first try to remove entire objects, and then for objects that remain visible, remove back-faces.

Stage V: There are a number of ways to compute how the objects would look through the lens of a virtual camera, but the process is greatly simplified when the camera is looking straight down the z-axis located at the origin. With this in mind, the world-to-camera transformation transforms all the geometry in the world as if the camera was at the origin looking down the z-axis. The transform is nothing more than a series of inverse transforms based on the translation and rotation transforms used to place the virtual camera.

Stage VI: Finally, with the camera at the origin (or similarly convenient position) with view angles all 0 (depending on convention), we have a viewing frustrum that represents what the camera's virtual lens can see. This frustrum usually has a near clipping plane and

Playrix Ex. 1023, page 565

09 0672318350 CH06 05•02•2003 2:40 PM Page 541

a far clipping plane, along with both a horizontal and vertical field of view. Objects within this frustrum are projected and rendered to the screen. This projection phase is the next part we are going to talk about.

Getting Ready for Projection

Before we talk about perspective coordinates, let's talk briefly about the entire projection process and the viewing frustrum once again. Assume that we have a 3D object made of a collection of vertices `v[]` that we want to render, and we have already performed all the transformations. The object is in the viewing frustrum as shown in Figure 6.18. The question is how to project this image onto the screen?

The answer is it depends on what you want to see. The first step is to look at the virtual camera system a little more so we can explain with the relationship between the camera, the world, and screen a little better.

Referring again to Figure 6.18, and assuming a 90 degree field of view system aligned to the z-axis and looking in the +z direction (a nice left handed system), this is the system we will use for the following discussion.

The Projection/Viewing Plane

The projection plane is where the action happens. Of course, the viewing frustrum itself determines what we process, but ultimately the viewing plane is used to form the virtual image of the 3D world, which is then used to display on the raster 2D screen. With our first camera system, we have decided that the camera will finally be positioned at location (0,0,0) looking straight down the +z-axis. With this setup, we can select the near, far, and view plane anywhere we like. However, the selection of the view plane will make some things easier or harder down the line in the viewing pipeline.

For example, because we are already assuming a 90 degree field of view, we can look at the 3D system from a top view without loss of generality, as shown in Figure 6.30, to arrive at some facts. These facts can then be generalized to full 3D. First, the distance that we place the view plane from the viewpoint (0,0,0) is sometimes called the viewing distance d. Now, the selection of d has a lot to do with the perspective transform and its relationship to the screen coordinates.

Now here's the clincher—after we define all these values and then place a 3D object in the frustrum, to render it onto the view plane we simply project mathematical *projectors* from each point that make up the object through the view plane that terminate at the viewpoint. This is all there is to projection! The image formed on the virtual view plane is then rendered on the screen of your computer.

Of course, there are some details we are missing here that have to do with scale, but we will get to that in a minute. At this point, let's take a closer look at the perspective transformation.

Playrix Ex. 1023, page 566

09 0672318350 CH06 05•02•2003 2:40 PM Page 542

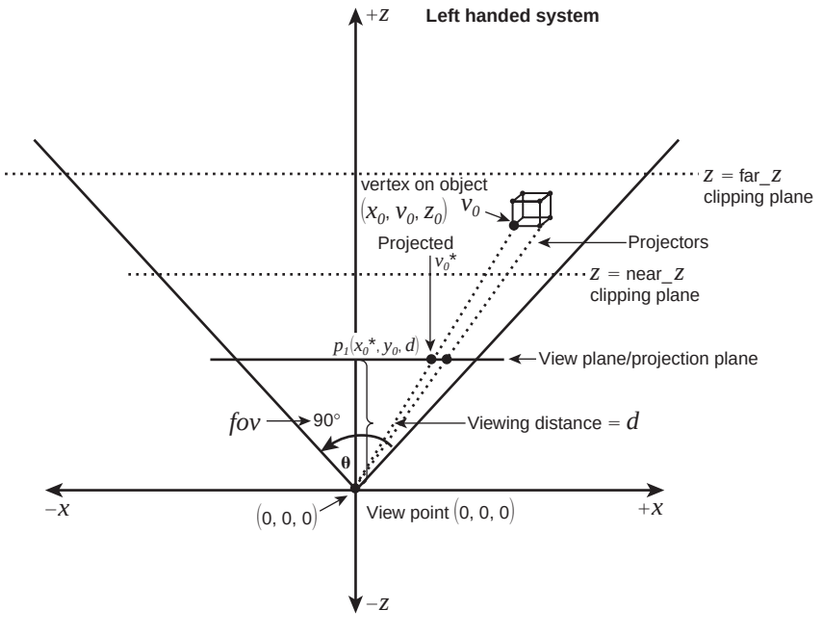


FIGURE 6.30 A simplified 2D top-down projection of the viewing frustrum in the x-z plane.

The Perspective Transformation

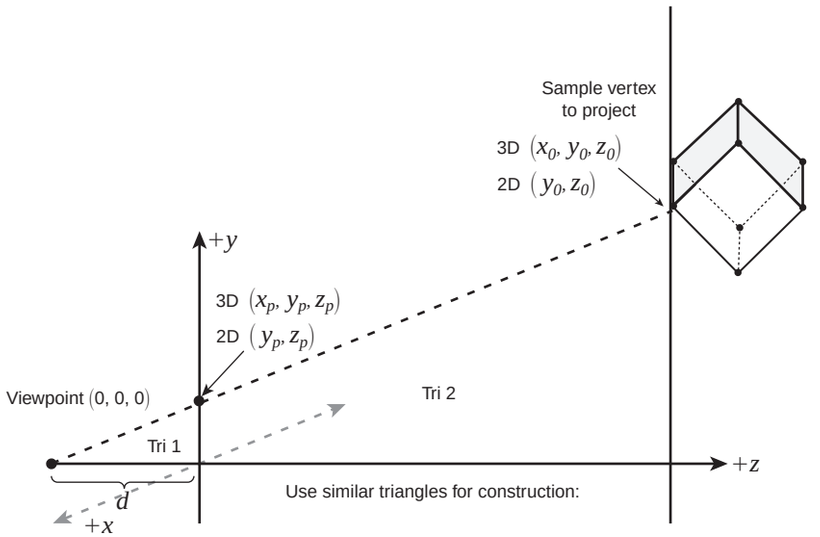
The perspective projection is nothing more than projecting an object's vertices through an arbitrary view plane, as shown in Figure 6.31. The formulation is as follows: Given that the view plane is a distance d from the point of projection (the camera viewpoint), then the math can easily be worked out to compute the intersection point in the view plane of a ray from the viewpoint through the view plane to a point on the object, or vice versa.

This geometry problem can be worked out using the 2D case and then generalized to the 3D case, so let's do that. In Figure 6.31, we see our 3D system on its side viewing the y-z plane. We also see a point **p**(y0,z0) that we want projected on our view plane, which is located a distance d from the origin at position z = d. The math is trivial, and nothing more than similar triangles that can be stated as follows: The side d is as to z as the y_p is to y. Thus, we can write the relationship as

$d/z0 = y_p/y0$

Playrix Ex. 1023, page 567

09 0672318350 CH06 05•02•2003 2:40 PM Page 543



$$xp = \frac{d \bullet x_0}{z_0} \qquad xp = \frac{d \bullet y_0}{z_0}$$

FIGURE 6.31 The construction of the perspective transform.

Rearranging this, we get

```
y_p = d*y0/z0
```

which means that the projected point y_p on the projection plane z = d from the projection point (0,0,0) (which is the camera, of course) is equal to the product of the distance from the camera d multiplied by y0, then divided by z0. Additionally, if we were to do the same analysis from the x-z plane, we would get the following relationship:

```
d/z0 = x_p/x0
```

Rearranging here, we get

```
x_p = d*x0/z0
```

Putting them together, we get the final perspective transformation for a view plane located at z = d from the viewpoint located at (0,0,0):

FORMULA 6.4 The Perspective Transformation

```
x_per = d*x/z
y_per = d*y/z
```

Playrix Ex. 1023, page 568

09 0672318350 CH06 05•02•2003 2:40 PM Page 544

First let's talk a little about problems with this formula. There's a *pole* at z = 0. That is, the equations go to infinity when z = 0, which is a bad thing. Alas, you can't project vertices with z = 0. Furthermore, objects with negative z values will get inverted, but still projected. That's why we need a near clipping plane, so this stuff can't happen.

NOTE

There's also a parallel or orthographic projection that is sometimes used in 3D for mechanical and CAD programs. In this projection, the z coordinates are simply thrown away. Of course, there's no perspective, but many times when designing parts and homes, you don't want to see any perspective. Thus, during the projection phase of a vertex (x0,y0,z0), the point is simply projected at (x0,y0), and sometimes z0 is used as an intensity value to shade the near imagery relative to z distance.

As always, let's take a look at the perspective transformation in terms of a code and a matrix formulation. The perspective transformation is very simple in terms of code. But we have to use homogenous coordinates—*period*! The reason is that there's no easy way to get a division by z with a matrix that doesn't contain z for each vertex, so hold that thought and look at how we would do the transform with brute force:

```
POINT3D cube_camera[8]; // used to hold camera coordinates
POINT3D cube_per[8]; // used to hold perspective coordinates

// perform the perspective transform on the camera
// coordinates, assume viewing distance is d
for (int vertex = 0; vertex<8; vertex++)
    {
    float z = cube_model[vertex].z;

    cube_per[vertex].x=d*cube_camera[vertex].x/z;
    cube_per[vertex].y=d*cube_camera[vertex].y/z;

    // we don't need this, so just copy it
    cube_per[vertex].z=cube_camera[vertex].z;
    } // end for
```

Brute force is good sometimes, isn't it? Hence, we can perform the perspective transform as shown and be done, but if we want to create a completely matrix-based system, we have to be able to represent everything with matrices. Therefore, here's the matrix T_{per} that performs the perspective transformation on 4D homogenous coordinates.

$$\mathbf{T}_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Playrix Ex. 1023, page 569

09 0672318350 CH06 05•02•2003 2:40 PM Page 545

Let's see whether it works with an example point in camera coordinates p = [x_c y_c z_c 1]:

p * $\mathbf{T}_{per}$ =

$$[x_c\ y_c\ z_c\ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

= [x_c, y_c, z_c, (z_c/d)]

Then if we divide the components (x,y,z) by w = (z_c/d) to convert from homogenous coordinates back to normal coordinates, we get the final point:

= [x_c*d/z_c, y_c*d/z_c, z_c*d/z_c]

And of course, we don't need to even look at the z coordinate, because after projection, we are only interested in the (x,y) coordinates, which are

$x = x_c*d/z_c$
$y = y_c*d/z_c$

This performs the perspective transformation via a matrix. However, this is the first time we've had to convert from 4D homogenous coordinates back to 3D. We might want to rethink whether we really need to perform the perspective transform using a matrix. The answer is that it depends on how we build the engine. That is, do we want a totally matrix-based system, or are we going to use matrices for the majority of the transformations, but in the quest for speed use simple code for operations like the perspective transformation and screen transformation? We won't worry about that right now, but it is an issue.

We're just about to slay the projection dragon, but there's still another detail we need to consider—the viewing distance d. What value should we pick for d? This is a good question, but you might not like the answer: anything! Yes, you can pick anything you like for d. Changing d will artificially change the scale of imagery that is the amount of perspective. So then, what's the best d? I usually pick d in one of two ways. First, if I want to really simplify my math, I select d = 1, which turns the perspective projection into the following:

FORMULA 6.5 Perspective Projection with d = 1 (Normalized Coordinates)

$x_{per} = 1*x/z$
$y_{per} = 1*y/z$

Playrix Ex. 1023, page 570

09 0672318350 CH06 05•02•2003 2:40 PM Page 546

The matrix for this operation is obtained by setting d = 1:

$$\mathbf{T}_{per1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

If you perform the preceding operation, you are going to end up with *normalized view plane coordinates*. In other words, all the vertices that are projected will fall in the range of –1 to +1 on both the x and y axes. Additionally, you need to map the view plane coordinates to screen coordinates for viewing. That is, map –1 to 1 on both the x and y axes to the screen resolution of width×height. There's a problem with this, though, because we are mapping a square view plane to the view port, and the view port must be square. Otherwise, one of the axes will end up being distorted, so in cases where the final view port or screen is nonsquare, we must take this into consideration and give the view plane the same ratio of width to height as the view port, or perform a secondary scaling on one of the axes at some point from the perspective-to-screen coordinates transformation. More on this later, but keep it in mind.

The other way to find d is to compute it based on the final screen coordinates that you want to generate. If you are going to project onto a screen that's 640×480, you can pick d such that the field of view is 90 on the x-axis, and all projections will lie in the range 640×480. However, there's a little problem here, because the x and y sizes are different. That means that you would think you would need a d for both the x-axis perspective transform and the y-axis transform. But if you use a different d for each, you will end up distorting the image on the screen. So in the case where you are projecting onto a screen of width×height and computing the d that gives a specific FOV for the x-axis, you MUST use that same d for both the x and y perspective transforms, and the end result will be that the y-axis field of view will be slightly different.

When you have a nonsquare projection view plane or view port, you need to be aware that the nice square projection won't work anymore, and that there's a factor that relates the x and y-axes together. This is called the *aspect ratio*. The aspect ratio is usually defined as width/height, so in the case of a 640×480 screen, the aspect is 4:3, or 1.33333. This number comes in handy during the perspective-to-screen projection for nonsquare view planes. In this case, you must multiple the aspect ratio times the focal distance during the perspective calculations.

Take a look at Figure 6.32 to see the construction and solution to find d. Then we simply use the same d for the vertical axis because we must use the same focal length.

NOTE

As a side effect, the vertical field of view will not in general be the same as the horizontal field of view, because the ratio of width/height is not 1.0.

Playrix Ex. 1023, page 571

09 0672318350 CH06 05•02•2003 2:40 PM Page 547

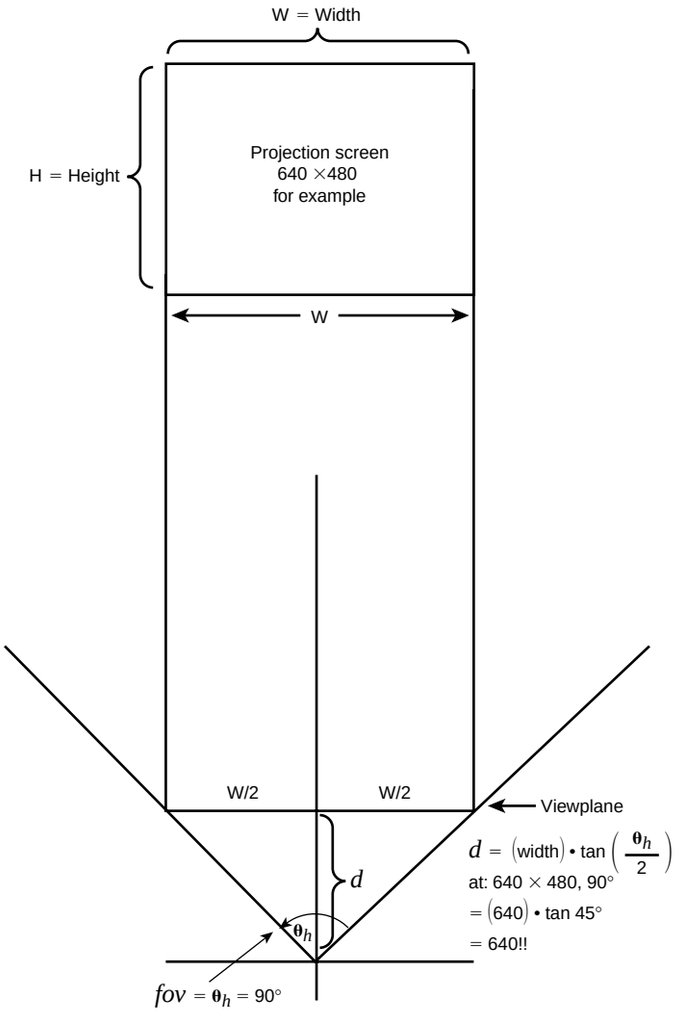


FIGURE 6.32 Computing d as a function of screen size and field of view.

FORMULA 6.6 Computation of Distance Values for Perspective Projection onto View Plane with Dimensions (Width×Height)

```
d = (0.5)*width*tan(θh/2)
```

Stated in plain English: The distance is equal to one half the width (or height, if you want that to be the major axis) of the projection plane times the tangent of the viewing angle divided by two. If you don't follow this, just look closely at the figure and remember that

Playrix Ex. 1023, page 572

09 0672318350 CH06 05•02•2003 2:40 PM Page 548

tan(θ) = opposite/adjacent = y/x, and in this case, y is the distance and x is related to the width.

Now that we have our distance factors for the projection onto a virtual view plane of size width×height, let's plug them back into the equations:

FORMULA 6.7a The Perspective Transformation onto a Square View Plane

$x_{per} = d*x/z$
$y_{per} = d*y/z$

To generalize this, we need to add in the aspect ratio factor ar. We can do this by multiplying it against the y component in the calculation—this generalizes the perspective transform:

FORMULA 6.7b The Perspective Transformation Generalized to Nonsquare View Planes

$x_{per} = d*x/z$
$y_{per} = d*y*ar/z$

And if you want to put it into matrix form the following matrix would perform desired transform:

$$\mathbf{T}_{perd} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d*ar & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

NOTE

Notice the repositioning of the d terms.

Given a point p $[x_c\ y_c\ z_c\ 1]$, let's multiply it against the matrix and check the results:

$$p * T_{perd} =$$

$$[xc\ yc\ zc\ 1] * \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d*ar & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= [(d*x_c)\ (d*ar*y_c)\ z_c\ z_c]$$

Playrix Ex. 1023, page 573

09 0672318350 CH06 05•02•2003 2:40 PM Page 549

Dividing the x,y components by the w component w = z_c, we get

$= [(d*x_c)/z_c \ (d*ar*y_c)/z_c \ 1]$

or

$x = (d*x_c)/z_c$
$y = (d*ar*y_c)/z_c$

which is identical to Formula 6.7b, and is indeed the transformation we were looking for!

The point is that you can do this stuff with either matrix or manual methods. You might ask, why use a matrix at all? The answer lies in the fact that matrices are much more convenient, and you can concatenate many matrix operations together: that is, $\mathbf{M} = \mathbf{M}_1*\mathbf{M}_2*\ldots\mathbf{M}_i*\ldots\mathbf{M}_n$. Thus, it's possible to concatenate local object transforms with the local-to-world, then world-to-camera, then camera-to-perspective, all in one matrix! The point being if you have to multiply any point by a matrix, the matrix multiplier doesn't know that you have 100 operations in the single matrix. That's the power of matrix multiplication, if you set it up right. But let's move on, keeping that in the back of your mind.

Perspective Projection and 3D Clipping

Simply stated, the perspective projection operates on the camera coordinates of the vertices of geometry that you want to display (which is determined in a previous step). The perspective projection T_{per} (or various derivatives that we have explored) basically transforms the points in camera coordinates within the viewing volume onto the view plane, readying them for display in the final step to screen coordinates, which we will get to in a moment. However, there is another use for the perspective transformation in the context of clipping: turning the pyramid shaped viewing volume into a *cuboid* shape.

In Figure 6.33, we have a simple 2D projection of what I'm talking about. Suppose that we have two points **p1**(x1,y1,z1) and **p2**(x2,y2,z2) that define a line. We haven't yet talked about 2D or 3D clipping, but let's talk about it now. Imagine that we have a single line defined by **p1** and **p2**, and that the line as it's shown in Figure 6.33 is partially in the viewing frustrum. Further assume that we have a wireframe 3D engine, so we need to draw this line. The line can't be removed with object removal because it really doesn't define an object, and the line doesn't have a back-face because it's not a polygon.

Basically we have to clip the line to the viewing frustrum to actually draw it. This step is very important, because all objects that are totally contained within the viewing frustrum can trivially be projected and rendered. Similarly, all objects that are entirely out of the viewing frustrum can be trivially rejected. However, geometry that is partially within the viewing frustrum must be clipped somehow, just as you would clip lines or bitmaps in a 2D window. Of course, I'm simplifying all this, and in real life, we are going to have to clip polygons to the viewing frustrum, but bear with me.

Playrix Ex. 1023, page 574

09 0672318350 CH06 05•02•2003 2:40 PM Page 550

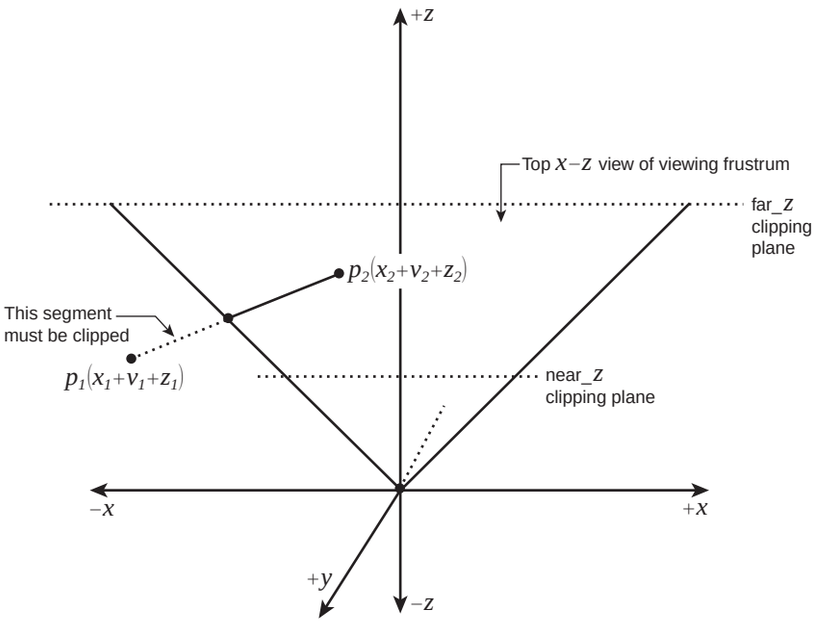


FIGURE 6.33 Simple clipping example.

There are two ways we can handle the line clipping—with the first method, we can blindly perform the perspective projection of the line (assuming that z > 0 on both end points), wait until we get to the rasterization phase of the 3D pipeline, and then clip the line in 2D against the window. This is called *image space clipping*. In other words, image space clipping involves waiting until all the objects to be rendered are already transformed into screen coordinates (although they might be out of range of the screen) and then clipping them in screen space or to the view port, as shown in Figure 6.34.

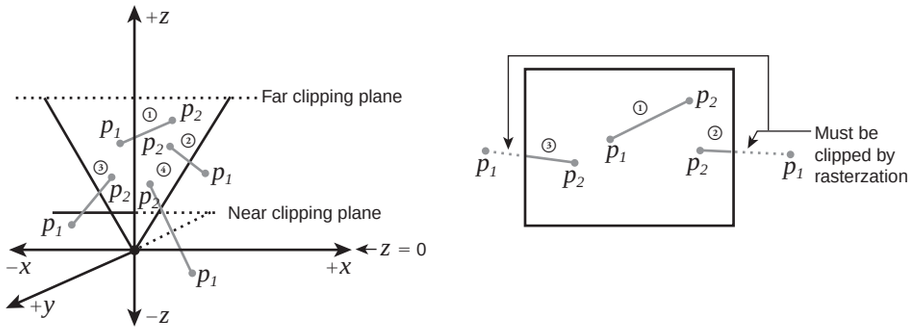
Image space clipping works great and is easy to implement, but it means that every single object gets clipped in 2D—even the objects that are entirely in the viewing window. This can be a great load on the processor, especially if you are rasterizing polygons and testing on a per pixel basis. But in the case of drawing lines, image space clipping in 2D is probably cheaper than clipping in 3D (which I'll explain in a minute).

The second way to clip is in *object space*; that is, in the mathematical space that the object(s) or geometry exist in. If you recall, we tried to remove entire objects from the pipeline and/or back-faces, so we didn't have to transform them to camera coordinates. This was an object space operation. However, some of the objects and geometry will of course be visible in a game (or you wouldn't have a game). Some of those objects will not be totally in the viewing volume, and hence must be clipped against it. Therefore, the problem boils down to this: Before you perform the perspective transformation on your

Playrix Ex. 1023, page 575

09 0672318350 CH06 05•02•2003 2:40 PM Page 551

geometry, if you want to perform object space clipping, all geometry (lines or polygons) that are only partially in the viewing frustrum must be clipped in 3D space to the viewing frustrum. This is called *3D clipping*, and it is not trivial unless you're Chris Knight.



Step 1: All lines (1, 2, 3) have z coordinates > 0 but line 4 does **not**, therefore projection of 4 will cause trouble.

Step 2: Project all lines (1, 2, 3) with perspective projection into screen space and clip during rasterzation.

FIGURE 6.34 The image space clipping pipeline.

In general, we have been dealing with a viewing frustrum that is fairly simple: There's a near and far clipping plane, the sides of the frustrum on the x and y axes are at 90 degrees FOV, and assuming a square projection plane, the plane equations are |x| = z and |y| = z, but still, clipping a 3D line to these six faces can be tedious. If the view plane isn't square, |x| = z, but |y| = z/ar.

Additionally, if the four side walls are not based on a 90 degree viewing angle, it's much harder, because the plane equations will be in general form.

The point is if you first clip all the geometry that is partially within the viewing frustrum to the viewing frustrum in 3D space (object space clipping) as shown in Figure 6.33, you are guaranteed that all the projected geometry will lie within the view plane. As I've mentioned, performing 3D clipping is a bit tricky, and when we get to it, you will see that not only do the polygon vertices have to be clipped, but the texture and lighting coordinates do, too—but more on that later in the book.

However, clipping to a cube or cuboid is much easier in 3D, because it's basically like clipping to a rectangle in each plane. Okay, now I can finally get to my point about the relationship between the *perspective transformation* and *clipping*. If you're willing to perform a perspective transformation on everything that is potentially visible, the perspective transform itself will warp the viewing volume into a cuboid, and the four sides that were at angles (90 degrees in our examples) will become parallel to each other on the x and y

Playrix Ex. 1023, page 576

09 0672318350 CH06 05•02•2003 2:40 PM Page 552

axes, and you will end up with a cube to which you can then clip your perspective coordinates. Amazing, huh? Let's see an example of this and see whether it's true.

Because anything we do to one plane works the same for the others, let's look at the top view of the x-z plane of the viewing frustrum, labeled as shown in Figure 6.35a, with a far clipping plane at z = far_z, a near clipping plane at z = near_z, and a projection, or view plane, at d = 1. The view angle is of course 90 degrees, and let's assume a square projection plan. What we want to do is perform the perspective projection on the corner points of the viewing pyramid (in the 2D case) and see what they look like after the transform. That is, what is the shape of the viewing frustrum itself if we were to perspective transform it?

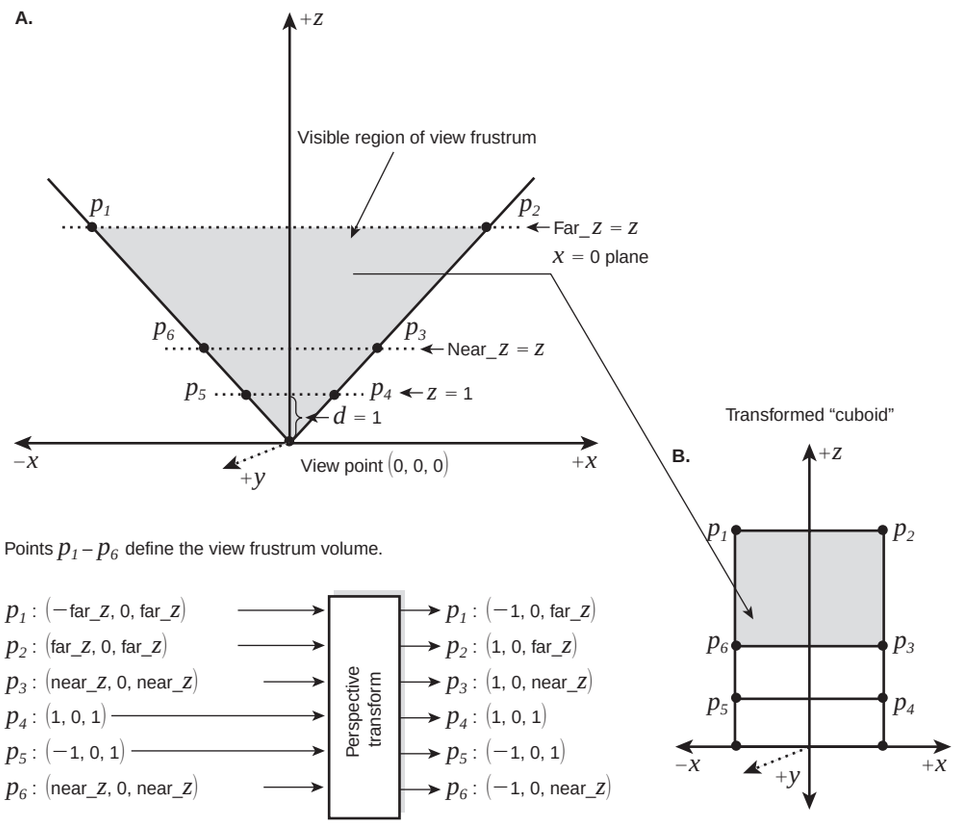


FIGURE 6.35 The perspective projection magically converts the viewing frustrum to a cuboid shape.

With that in mind, the points that make up the 2D projection on the x-z plane of the viewing volume are **p1**–**p6**, and they have the following actual coordinates in the form **p**(x,0,z):

Playrix Ex. 1023, page 577

09 0672318350 CH06 05•02•2003 2:40 PM Page 553


```
p1(-far_z, 0, far_z)
p2(far_z, 0, far_z)
p3(near_z, 0, near_z)
p4(1,0,1) ß- note the x components are 1 since d=1
p5(-1,0,1)
p6(-near_z, 0, near_z)
```

TIP

You might be concerned that the x components of some of the points are in terms of z. This is because we are using a 90 degree field of view with a square projection plane; thus |x| = z, and |y| = z, so at any z value from the origin, {x,y} = z.

Now watch this baby! If we perform the perspective projection on these points with d = 1 (that is, multiply each component by d), and then divide by z, we get this:

```
p1'(-far_z/far_z, 0/far_z, far_z)     = (-1, 0, far_z)
p2'(far_z/far_z, 0/far_z, far_z)      = (1, 0, far_z)
p3'(near_z/near_z, 0/near_z, near_z)  = (1, 0, near_z)
p4'(1/1, 0/1, 1)                      = (1, 0, 1)
p5'(-1/1, 0/1, 1)                     = (-1, 0, 1)
p6'(-near_z/near_z, 0/near_z, near_z) = (-1, 0, near_z)
```

Of course, we can ignore the y components, because we know we are in the x-z plane.

Now, let's re-plot **p1'**–**p6'** and take a look at Figure 6.35b. Look, it's a rectangle. Aha! We have transformed the ugly pyramid into a nice clean rectangle, which is trivial to clip against, even in 3D. Of course, we can perform the same analysis on the y-z plane and come up with the same results. Merging them together, we get the new cuboid viewing volume shown in Figure 6.35b. The point of this exercise is to show you that the perspective transform can also be used to *normalize* all the geometry, so you can clip it trivially. Of course, you will see this in action in the upcoming chapters.

In conclusion, our 3D pipeline has more than just a perspective transform added onto it. The pipeline has a possible 3D clipping phase (if you don't want to clip in image space), so our new pipeline with 3D clipping up to the perspective transform is shown in Figure 6.36. Additionally, if we decide to transform all geometry from camera coordinates through the perspective transform, we can clip in 3D after the perspective transform, also shown in Figure 6.36—it's up to you how you want to do this.

Finally, let's get to the last step in the viewing pipeline—screen coordinates.

Playrix Ex. 1023, page 578

09 0672318350 CH06 05•02•2003 2:40 PM Page 554

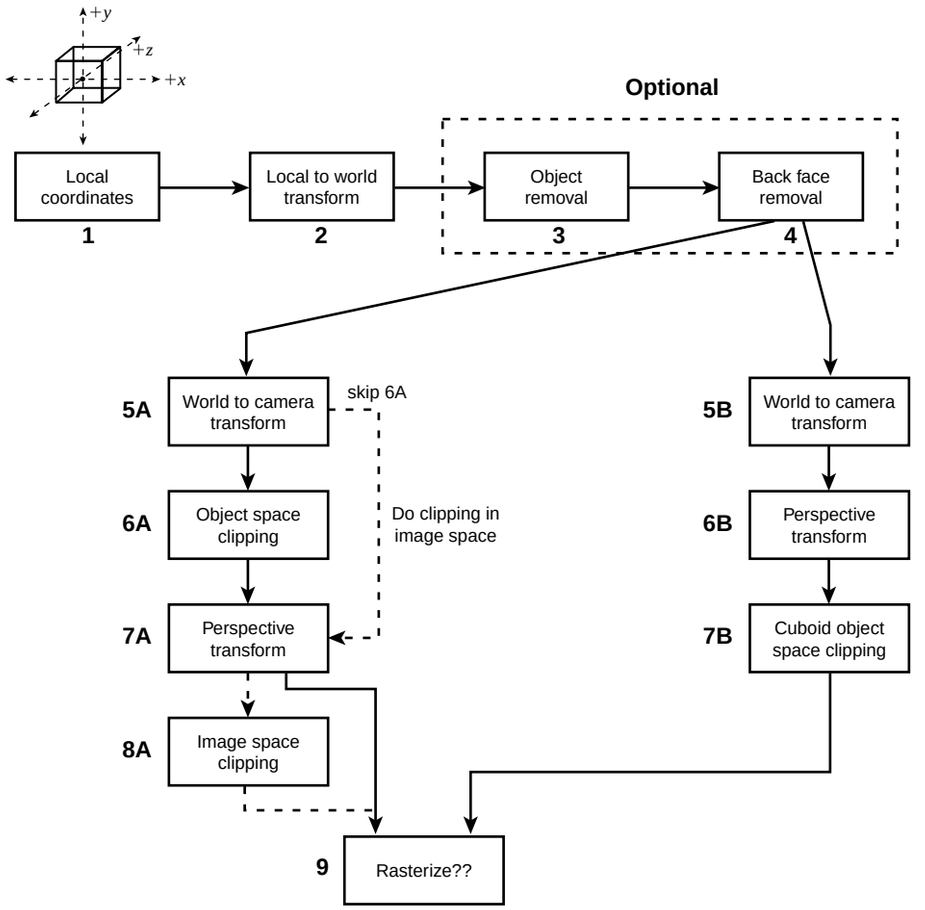


FIGURE 6.36 The next iteration of the 3D pipeline with the perspective transform and clipping.

The End of the Pipeline: Screen Coordinates

Before I describe the screen coordinate transform component of the pipeline, let it be known that we are still leaving out the entire lighting and texturing phases of the 3D pipeline. These are advanced topics and not appropriate for this chapter, but at least make note that somewhere between the world and screen coordinates we must light and texture our polygons. But for now, let's keep it simple, because I'm trying to stick to coordinate transforms.

Playrix Ex. 1023, page 579

09 0672318350 CH06 05•02•2003 2:40 PM Page 555

At this point in the pipeline, we have a set of points that represent what we want to see on the screen. These points have already passed through the perspective transform, but may or may not be clipped to the final screen dimensions. For now, let's just assume that we don't care and come back to it. However, computing the final screen coordinates does rely on the type of perspective transform that took place; that is, what was the size of the view plane in virtual space? This matters because we need to map it to the screen. So let's look at some of the cases.

Field of View 90 Degrees with View Plane at d = 1

With a field of view of 90 degrees and the view plane at a distance d = 1 with a square projection plane, the perspective transform transforms all geometry and projects it onto a virtual view plane of width and height 2×2 (x:{-1 to 1} X y:{-1 to 1}). In the case of a nonsquare projection plane (the more common), the perspective projection projects everything onto a virtual view plane of width and height 2×(2/ar) (x:{-1 to 1} X y:{-1/ar to 1/ar}), where ar is of course the aspect ratio. The cool thing is when we get to the screen transform, we don't care about this anymore, because we assume that the perspective transform handled the aspect ratio. However, it's VERY important that somewhere the aspect ratio is taken into consideration—otherwise, your imagery will be squished!

In any case, the projections can be verified for your sanity by viewing Figure 6.37. So, we want to map coordinates from the view plane, which is 2×2 for square projections and 2×(2/ar) for nonsquare projections, to our view port or screen coordinates. Read that sentence once more—we want to map the view plane coordinates onto the view port. Remember that the view plane is in virtual camera space, and may or may not have the same dimensions as the view port, which is the final raster window.

This transformation step can be interpreted in different ways, but the way we are going to assume that things work for now is that we render into a view port buffer with origin (0,0) at the upper left-hand corner with dimensions (`SCREEN_WIDTH` × `SCREEN_HEIGHT`), as shown in Figure 6.38. In this case, the width and height are equal. This is the exact setup of the virtual computer system we designed back in Chapters 2 and 3.

With this in mind, we merely have to come up with the following transform from view plane coordinates to screen coordinates (which is our *view port*):

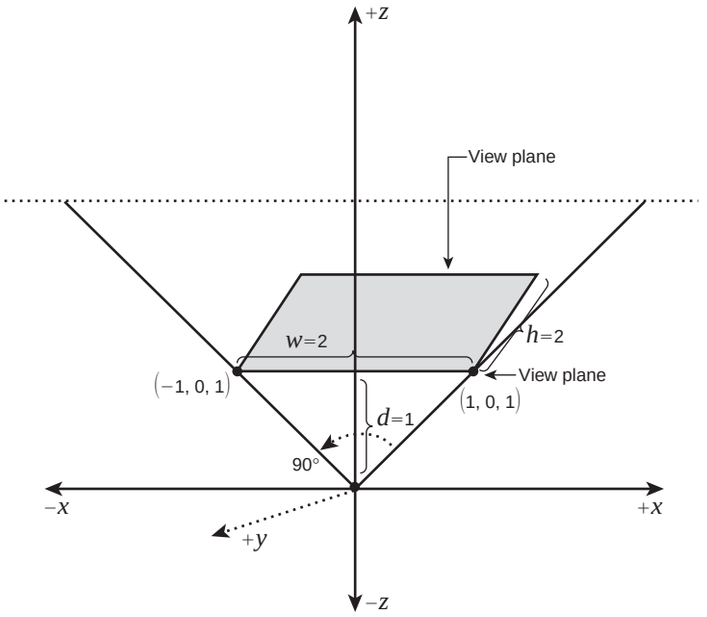
x_{per}: (-1 to 1) -> x_{screen}: (0 to SCREEN_WIDTH-1)
y_{per}: (-1/ar to 1/ar) -> y_{screen}: (SCREEN_HEIGHT-1 to 0)

NOTE

The y-axis is inverted, and details count here, boys and girls—we can't afford to be "off by one;" we need to be perfect!

Playrix Ex. 1023, page 580

09 0672318350 CH06 05•02•2003 2:40 PM Page 556



When the fov = 90 and aspect ratio = 1.0
then view plane = (2×2) d = 1

When the fov = 90 and aspect ratio ≠ 1.0
then view plane = $(2\times2/\text{ar})$ d = 1

FIGURE 6.37 Projection details for FOV = 90 and d = 1 onto 2×2 projection plane.

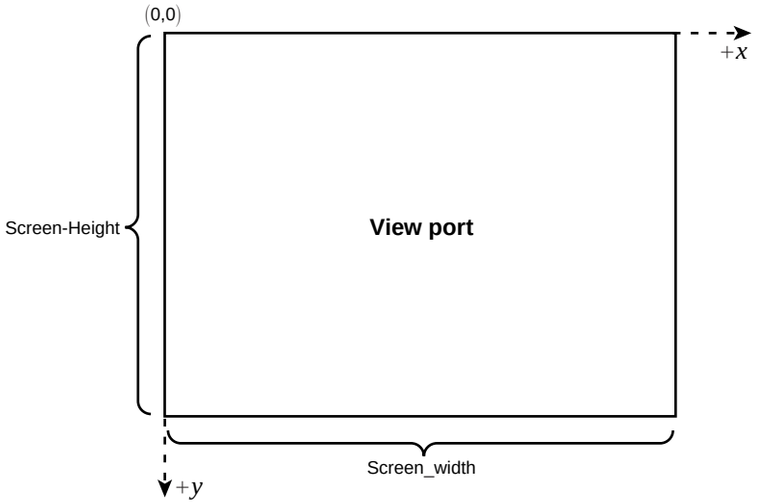


FIGURE 6.38 The viewport definition.

Playrix Ex. 1023, page 581

09 0672318350 CH06 05•02•2003 2:40 PM Page 557

There are a number of ways to derive the math here, but the easiest is to find the center of the screen and transform from there. However, there is one thing to remember: The y-axis is inverted, so we need to take that into consideration. With that in mind, to transform the point **p**(x,y) to screen coordinates, we use the following transform:

FORMULA 6.7 Conversion to Screen Coordinates for d = 1 Case

$$x_{screen} = (x_{per}+1)*(0.5*SCREEN\_WIDTH-0.5)$$
$$y_{screen} = (SCREEN\_HEIGHT-1) - (y_{per}+1)*(0.5*SCREEN\_HEIGHT-0.5)$$

Let's test this with some boundary conditions, like the four points that make up the bounding box of the view plane and transform them to screen coordinates. Performing this operation with Formula 6.7 we get

```
p1(-1,1)     = p1'(0, 0)
p2(1,1)      = p2'(SCREEN_WIDTH-1, 0)
p3(1,-1)     = p3'(SCREEN_WIDTH-1, SCREEN_HEIGHT-1)
p4(-1,-1)    = p4'(0, SCREEN_HEIGHT-1)
```

which is exactly the bounding box of the addressable screen with extents (SCREEN_WIDTH × SCREEN_HEIGHT). Of course, there are a number of ways that you can formulate the screen transformation by multiplying out the terms and collecting them in different ways. For example:

$$x_{screen} = (x_{per}+1)*(0.5*SCREEN\_WIDTH-0.5)$$
$$= (x_{per})*(0.5*SCREEN\_WIDTH-0.5) + (0.5*SCREEN\_WIDTH-0.5)$$

And if we let α = (0.5*SCREEN_WIDTH-0.5), we get

$$x_{screen} = \alpha + xper*\alpha$$

$$y_{screen} = (SCREEN\_HEIGHT-1)-(yper+1)*(0.5*SCREEN\_HEIGHT-0.5)$$

$$= -(y_{per})*(0.5*SCREEN\_HEIGHT-0.5) - (0.5*SCREEN\_HEIGHT-0.5) + (SCREEN\_HEIGHT-1)$$

$$= -(y_{per})* (0.5*SCREEN\_HEIGHT-0.5) + (0.5*SCREEN\_HEIGHT-0.5)$$

If we let β = (0.5*SCREEN_HEIGHT-0.5), we get

$$y_{screen} = \beta - yper*\beta$$

Playrix Ex. 1023, page 582

09 0672318350 CH06 05•02•2003 2:40 PM Page 558

Thus, we have the following final results:

$x_{screen} = \alpha + x_{per}*\alpha$
$y_{screen} = \beta - y_{per}*\beta$

where α = (`0.5*SCREEN_WIDTH-0.5`), and β = (`0.5*SCREEN_HEIGHT-0.5`).

What's interesting is that (α,β) is the exact center of our view screen (`SCREEN_WIDTH` × `SCREEN_HEIGHT`), which makes sense because we are mapping (0,0) there. The cool thing about the preceding formulation is that it's much easier to put in matrix format. However, we are at the point were we are really dealing with 2D screen coordinates, and using 4D homogenous coordinates isn't necessary. That is, if our perspective points are in the form (x,y), there's no point in using a 4×4 matrix. You could use a 3×3 matrix if you want. Sometimes it's nice to just keep everything in 3×3 or 4×4 matrices and disregard the z component at some point—whatever you decide is up to you, but I will show you both just to be complete.

With that in mind, let's look at a 4×4 version of the transformation, assuming that the view plane coordinates in perspective are in the form [x_{per}, y_{per}, z, 1], where z is irrelevant and could be zero because it's already served its usefulness. Because this is the perspective-to-screen transform for viewing distance d = 1, we will call the matrix T_{scr1}:

$$\mathbf{T}_{scr1} = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$

And if we multiply a test point p = [x_{per} y_{per} 0 1], we get

$\mathbf{p}* \mathbf{T}_{scr1} =$

$$[x_{per}\ y_{per}\ 0\ 1] * \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$

$[x_{per}*\alpha + \alpha,\ -y_{per}*\beta + \beta,\ 0,\ 1]$

And picking off the (x,y) components we get:

$x = x_{per}*\alpha + \alpha$
$y = -y_{per}*\beta + \beta$

which is correct! Of course, that was a lot of excess drama, because we could just use the preceding formula and forget the matrix transform! Remember: the idea is that at some

Playrix Ex. 1023, page 583

09 0672318350 CH06 05•02•2003 2:40 PM Page 559

point, we might be able to do 50 different things in a single matrix by concatenation, so that's why we're learning how to do everything in a matrix. Let's move on to a second example, assuming that we have a single 2D point (x_{per}, y_{per}) that represents the point that we want to transform into screen coordinates with a matrix, as we did previously. The problem is that we need to get a translation by (α,β), so we need homogenous coordinates. What we're going to do is use homogenous 2D coordinates—which will look like 3D coordinates, but with an (x, y, w)—then we can use a 3×3 matrix to perform the screen transform. With that in mind, let's say we want to transform the point p (x_{per}, y_{per}, 1) to screen coordinates, with d = 1 as usual. This matrix would do the trick:

$$\mathbf{T}_{scr1} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & -\beta & 0 \\ \alpha & \beta & 1 \end{bmatrix}$$

And if we multiply a test point p = [x_{per} y_{per} 1], we get

$$\mathbf{p} * \mathbf{T}_{scr1}$$

$$[x_{per} \; y_{per} \; 1] * \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ \alpha & \beta & 1 \end{bmatrix}$$

$$[x_{per}*\alpha + \alpha, \; -y_{per}*\beta + \beta, \; 1]$$

Picking off the (x,y) components, we get

```
x = xper*α + α
y = -yper*β + β
```

which is again correct! My point is that you can do this stuff in a number of ways. Finally, let's look at some code to perform this operation, assuming that the perspective coordinates are already in the array `cube_per[8]`:

```
POINT3D cube_per[8]; // used to hold perspective coordinates

POINT2D cube_screen[8]; used to hold screen coordinates
// perform the perspective to screen transform
// on the perspective coordinates, assume screen has
// dimensions SCREEN_WIDTH x SCREEN_HEIGHT and
// view plane is at d=1 so perspective coords are
// normalized to -1 to 1 on each axis

// pre-compute alpha and beta
alpha = (0.5*SCREEN_WIDTH-0.5);
beta  = (0.5*SCREEN_HEIGHT-0.5);
```

Playrix Ex. 1023, page 584

09 0672318350 CH06 05•02•2003 2:40 PM Page 560


```
// loop and transform
for (int vertex = 0; vertex<8; vertex++)
    {
    cube_screen[vertex].x=alpha+alpha*cube_per[vertex].x;
    cube_screen[vertex].y=beta-beta*cube_per[vertex].y;
    } // end for
```

As you can see, doing the transformation manually without matrices is a lot less complex.

NOTE

Although I am being cavalier with the definitions of view plane, viewport, and screen coordinates, remember that the view plane is what we initially project on in virtual space, and the viewport is the window that we map the view plane coordinates on (which in many cases is the exact same as the screen window).

View Plane with Arbitrary Field of View and General Viewing Distance d

Now that you've seen the construction of the perspective-to-screen transformation where d = 1, and the field of view is 90 degrees where the view plane coordinates in perspective space are normalized to –1 to 1 (on each axis for square projections), let's talk about the more general case, where we compute the d's like this:

$d = (0.5)*viewplane\_width*\tan(\theta_h/2)$

Remember, in this case we solve for d based on the major axis (usually the x) field of view angle: θ_h (which would normally be 90 degrees for a square projection plane, but could be anything in this case). Additionally, we need the view plane dimensions we want to project the final image on, which would be (SCREEN_WIDTH × SCREEN_HEIGHT) in case we want the view plane to be the same size as the screen or view port. However, we need to be careful, because screen coordinates have the following ranges:

x: (0..SCREEN_WIDTH-1)

y: (0..SCREEN_HEIGHT-1)

So, in the previous formulas, we must replace viewplane_width and viewplane_height with the slightly different ranges (SCREEN_WIDTH-1) and (SCREEN_HEIGHT-1), like this:

$d = (0.5)*(SCREEN\_WIDTH-1)*\tan(\theta_h/2)$

This calculation will give us the correct d value if we want to merge the perspective transform *and* screen transform into one. However, the SCREEN_HEIGHT seems to have been left out—NO! Normally, we first project the image onto the view plane with a focal distance d

Playrix Ex. 1023, page 585

09 0672318350 CH06 05•02•2003 2:40 PM Page 561

for both the x and y axes, with the view plane size {x:-1 to 1, y:-1/ar to +1/ar}. Then we would map that onto the screen by multiplying by the SCREEN_WIDTH and SCREEN_HEIGHT on the x and y axes. But, because the ratio of SCREEN_WIDTH/SCREEN_HEIGHT is exactly the aspect ratio ar, the multiplication and division cancel out. Thus, when you map directly to the screen without the intermediate step of first going to view plane coordinates and then to viewport screen coordinates, the aspect ratio terms cancel out, and you are left with

$x_{per} = (d*x_c)/z_c$
$y_{per} = (d*y_c)/z_c$

The results will already be scaled to (SCREEN_WIDTH × SCREEN_HEIGHT), so you get the screen transformation free during the perspective transformation! Well, almost. There are a couple problems with this … Give up? First, the y-axis is inverted for raster display, because 0 is at the top of a raster display and at the bottom of quadrant I (which is where the projection is working). That's no problem: We can simply invert the y coordinates:

$y_{per} = (SCREEN\_HEIGHT - 1) - y_{per}$
$x_{per} = x_{per}$

The second problem is a bit more subtle, and much worse. Although the projection will have a total range of (SCREEN_WIDTH-1) on the x-axis and (SCREEN_HEIGHT-1) on the y-axis, the actual coordinates pumped out will be in the range of

X-axis: -(SCREEN_WIDTH-1)/2 to (SCREEN_WIDTH-1)/2

Y-axis: -(SCREEN_HEIGHT-1)/2 to (SCREEN_HEIGHT-1)/2

Centered about (0,0), the range is split into a positive and negative half about zero on each axis! Just as the normalized projection gave us coordinates –1 to 1 on each axis, we now have similar, although scaled, coordinates. Alas, although we've gotten rid of our post-perspective normalized-to-screen transform scaling operation, it looks like we are still stuck with some translation of the origin. This is true, but we can fix it, along with the y-axis inversion, with this simple transform after the initial perspective transform:

$x_{screen} = x_{per} + (0.5*SCREEN\_WIDTH-0.5)$
$y_{screen} = -y_{per} + (0.5*SCREEN\_HEIGHT-0.5)$

If you analyze the preceding operations, you will come to the conclusion that they are almost identical to the normalized screen projection *without* the scaling operation. So, as you can see, this projection stuff all ends up being the same thing in the end—it's just how you want to think about it. In fact, if we let a = (0.5*SCREEN_WIDTH-0.5) and b = (0.5*SCREEN_HEIGHT-0.5) as before, we have Formula 6.8:

Playrix Ex. 1023, page 586

09 0672318350 CH06 05•02•2003 2:40 PM Page 562

FORMULA 6.8 Screen Projection from View Plane Scaled Coordinates to Viewport Coordinates

$$x_{screen} = x_{per} + \alpha$$
$$y_{screen} = -y_{per} + \beta$$

And in matrix form, we have only to perform the translation to (x_{per}, y_{per}). Thus the transform, called $\mathbf{T}_{scr}$, would be

$$\mathbf{T}_{scr} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$

The preceding transform only differs from our more complex normalized-to-screen coordinate transform shown in the following by only the scaling factors (α,β) multiplied by x_{per} and y_{per}, respectively, which does nothing more than scale the view plane to a size $(|2*\alpha| \times |2*\beta/ar|)$, because the actual coordinate range is from x: $(-\alpha$ to $\alpha)$ and y: $(-\beta$ to $\beta)$:

$$x_{screen} = \alpha + x_{per}*\alpha$$
$$y_{screen} = \beta - y_{per}*\beta$$

The point is that we can scale the view plane coordinates before the screen transform as part of the perspective transform (but this gives us non-normalized coordinates), or we can scale the normalized coordinates to screen (view port) coordinates during the perspective-to-screen transformation.

Finally, we come to my last point—we have seen how to perform the perspective transform in its two flavors: normalized and view plane scaled to viewport, along with the two screen transforms that handle these previous types of perspective projections. But there's one little detail that I want to bring up—the conversion from homogenous 4D coordinates to 3D in the middle of the process. We have assumed that we are breaking the perspective transform and the screen transform up into two matrix multiplies with a quick homogenous-to-3D conversion of each vertex between the two transforms, but what if we want to perform the entire camera-to-screen transform all at once, both manually and with a single matrix? Let's look at it manually first.

Basically, we would perform the perspective transform along with scaling and translation all at once, and the math would look like this:

$$x_{screen} = d*x_{cam}/z_{cam} + (0.5*SCREEN\_WIDTH-0.5)$$
$$y_{screen} = -d*ycam/z_{cam} + (0.5*SCREEN\_HEIGHT-0.5)$$

Amazingly trivial, huh? Anyway, the simplicity of the preceding math is why many engines (mostly software) don't use matrix-based perspective and screen transforms. After

Playrix Ex. 1023, page 587

09 0672318350 CH06 05•02•2003 2:40 PM Page 563

vertices are in camera coordinates, the preceding two lines of code are used to transform all the way from camera coordinates to screen coordinates. But if we want to use a single matrix operation, it's a bit tricky because we have to assume 4D homogeneous coordinates, and that there will be a final conversion from homogenous to 3D (actually 2D) at the end via the division by w. With that in mind, take a look at the following matrix, which is a first attempt:

$$\mathbf{T}_{camscr} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \alpha & \beta & 0 & 0 \end{bmatrix}$$

That looks about right. Let's try it out with a vertex in camera coordinates $p = [x_c\ y_c\ z_c\ 1]$:

$$\mathbf{p}*\mathbf{T}_{camscr} = [d*x_c + \alpha,\ -d*y_c + \beta,\ z_c,\ z_c]$$

Although this looks right, when we convert from 4D homogenous coordinates to 3D, we divide x,y by $w = z_c$ and get

$$x_{screen} = d*x_c/z_c + [\ \alpha/z_c\ ] \text{ <- last terms are wrong}$$
$$y_{screen} = -d*y_c/z_c + [\ \beta/z_c\ ]$$

which is almost correct, but during the 4D to 3D conversion, we had to divide everything by $w = z_c$, resulting in the α and β terms being wrong (see the factor of z_c in the denominator of each). So what we need to do is scale them by z_c in the initial matrix, so that during the final conversion from homogenous coordinates to 3D, we get what we want. We can accomplish this by moving the α and β factors in the matrix up a row:

$$Tcamscr = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ \alpha & \beta & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

If we perform the same multiplication of the camera vertex $p = [x_c\ y_c\ z_c\ 1]$, we get

$$\mathbf{p}*\mathbf{T}_{camscr} = [d*x_c + zc*\alpha,\ -d*y_c + zc*\beta,\ z_c,\ z_c]$$

which, after dividing by $w = z_c$, gives us our desired transformation:

$$x_{screen} = d*x_c/z_c + \alpha$$
$$y_{screen} = -d*y_c/z_c + \beta$$

Playrix Ex. 1023, page 588

09 0672318350 CH06 05•02•2003 2:40 PM Page 564

CAUTION

The two things to remember about using the matrix-based perspective and screen transforms are that the resulting coordinates are homogenous out of the perspective transform and must be converted to 3D, and there are a lot of matrix multiplications and additions that are a complete waste of time.

Finally, you might ask, "Where did the aspect ratio factor go?" The answer is that is has been canceled out by combining the perspective-to-view plane and view plane-to-viewport (screen coordinate) transforms.

So, the question is, what methods should we use for both projection and the math itself? Well, they all have their place: normalized or scaled to the view port perspective transform, manual math, or matrix transformations, and so forth. In the exercises and demos in the next chapter, we will experiment with all of them, but in the end I think we will end up with the d = 1 with the FOV equal to 90 degrees on the x-axis formulation, because I like the normalized coordinates. However, later on we might find that merging the screen transform with the perspective transform is faster in real code.

Furthermore, as you can see we have 3–4 degrees of freedom with the view angle, focal distance, and the width and height. However, in most cases we *really* want to stick to FOV values that equal 90 degrees for the x-axis and let the math compute the FOV of the vertical axis, but in any case, use the same focal distance for the projection on both axes. However, for nonsquare projections, there must be a multiplication of the aspect ratio somewhere in there.

In conclusion, you might think that I have really beat down the whole projection process, but my goal is for you to really understand the relationship between camera coordinates, perspective coordinates, the view plane, the view port, and screen coordinates, as well as all the ways of arriving at them with various transformation sequences, both manual and matrix-based. The bottom line is that in this book, in most cases we will use a horizontal field of view of 90 degrees, and a focal distance of 1.0. That same focal distance will be used on the y-axis as well, so for nonsquare projections we will get distortion, but if we multiply the y component during the perspective calculation by the aspect ratio, it will cancel out when we scale by the `SCREEN_HEIGHT` in the screen mapping.

Another Look at the Viewing Pipeline

Now we have a complete picture of the viewing pipeline as far as geometry is concerned. Granted, we aren't doing lighting, shading, or texturing, but that really is a different part of 3D, under the topic of rendering, and we are only interested in the geometry transformations of the pipeline at this point. With that in mind, take a look at Figure 6.39, which depicts our system thus far.

Playrix Ex. 1023, page 589

09 0672318350 CH06 05•02•2003 2:40 PM Page 565

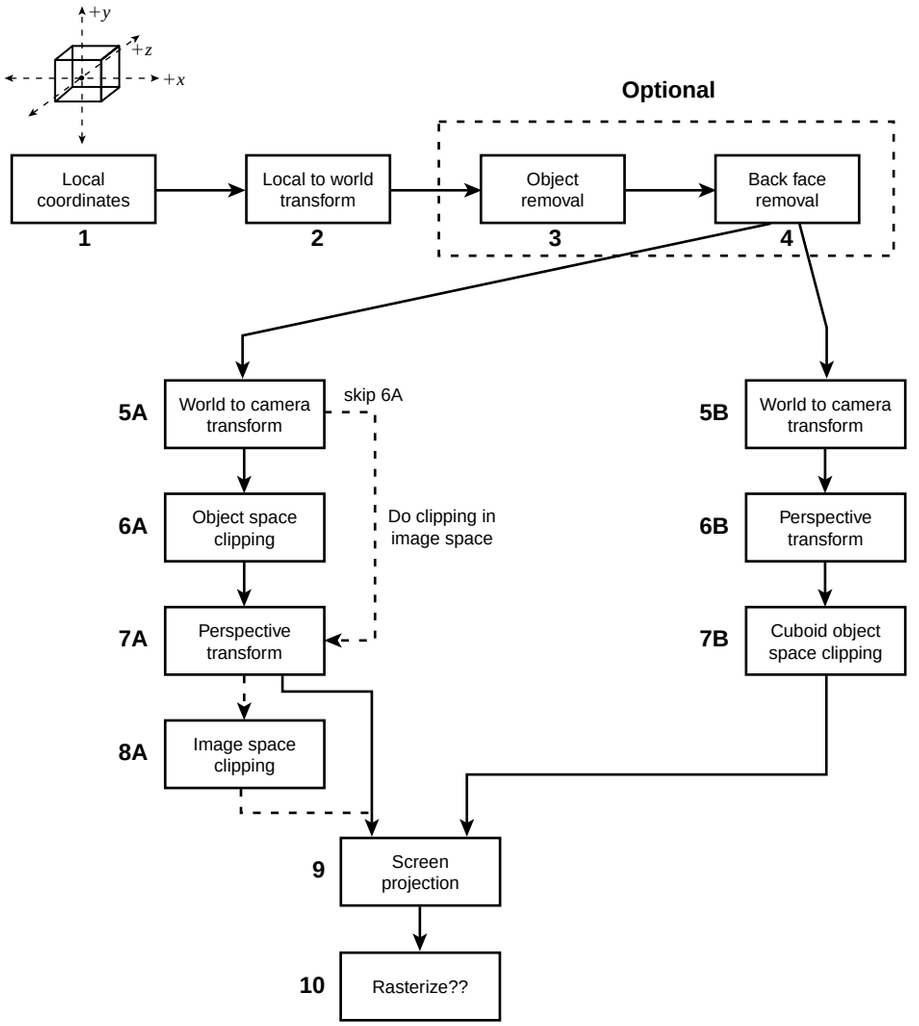


FIGURE 6.39 The next iteration of the 3D pipeline with the screen projection step added.

Basic 3D Data Structures

At this point, we are ready to start talking about general methods to represent 3D data. I think that you see there are a number of transformations that 3D data must go through, and there are a number of different representations as the data flows. Hence, there is no single correct way to do something—just the way that works in some specific context. For example, the way you store your data for a space game will probably be much different than that of an indoor game. Moreover, depending on how you clip your imagery, vertices

Playrix Ex. 1023, page 590

09 0672318350 CH06 05•02•2003 2:40 PM Page 566

might have to be inserted or deleted from your final rendered polygons, which takes some forethought. My buddy John Amato found this out writing his first 3D engine "SOL."

The bottom line is you need to really think out your engine, and many times you will end up redoing all of it because you might find that there's a better way to implement something. The approach I'm going to take in this book is a generic one. We will use some rough data structures to begin, and then tweak them as necessary. Again, the proper selection of data structures only comes after you know everything about 3D, but that won't happen until the end of the book, so at this point, we are just going to start off with something that works, along with some simplifications and assumptions.

My goal by the next chapter is to create a working wireframe 3D engine that enables you to move a camera around a space or terrain-like environment. The engine will do minimal clipping, and will probably be a hybrid of object space and image space algorithms. What I want to do now is discuss a couple of basic data structures that will help us represent 3D objects with a very simple generic 3D system in mind.

Considerations of Representing Polygonal Data in 3D

The first decision you have to make when designing a 3D engine is if you are going to support triangles, quadrilaterals, or general *n*-sided polygons, as shown in Figure 6.40. The majority of 3D engines (both software and hardware) only deal with triangles. The reasons are many, but most importantly, triangles are easy to render—three points define a plane, and many 3D algorithms work best with triangular meshes.

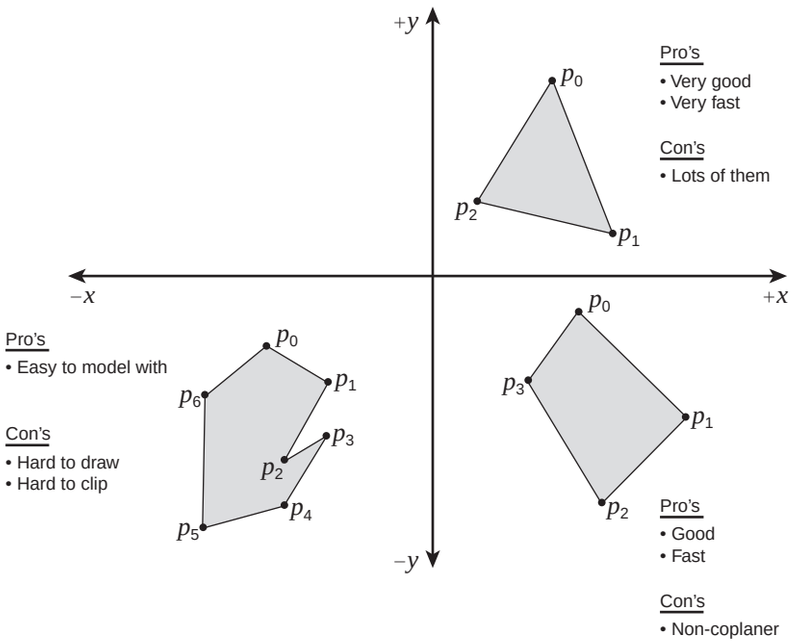


FIGURE 6.40 Potential polygonal primitives for a 3D engine to use.

Playrix Ex. 1023, page 591

09 0672318350 CH06 05•02•2003 2:40 PM Page 567

Of course, your models in the modeling tool might have general *n*-sided polygons, but at some point everything is triangulated. Because using triangles is the norm, we are going to go along with this convention and have the rule that all models must be built of three-sided polygons or converted to triangles before we render them. At this point, we're not sure where we are going to get the data for the objects. Is it algorithmically generated, entered by hand, loaded from files, and so on? And we don't care—we just know that we are going to deal with triangular-based geometry.

Even with this constraint there's a problem. Take a look at Figure 6.41, which shows a number of triangles being clipped to a rectangular region. In essence, during the clipping phase we can introduce more vertices into each polygon. For example, in the worst case, we might insert (add) three new vertices to any triangle after it's clipped. Hence, we have to have data structures that can deal with this.

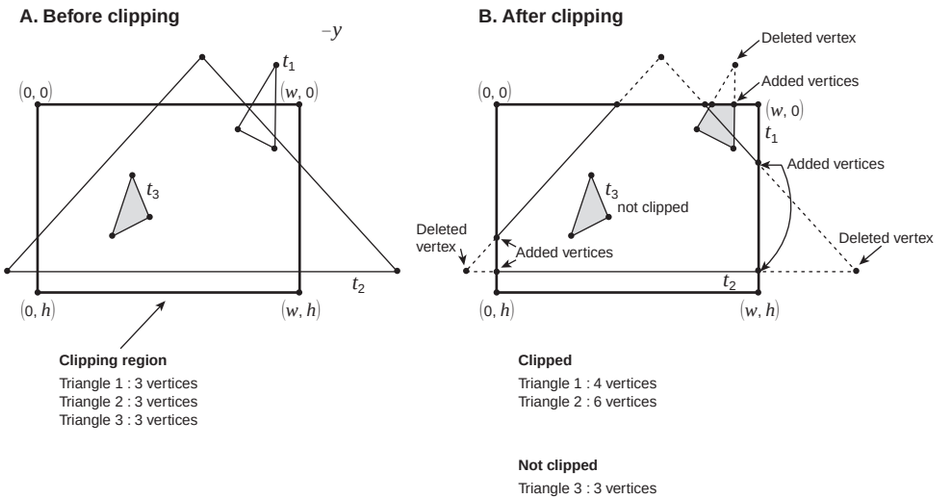


FIGURE 6.41 Triangle clipping in object space introducing extra vertices.

Now, you could say that we aren't going to clip at all in object space, but wait until the rasterization phase. This is fine, but then you run into the problem of polygons that pierce the near clipping plane, as shown in Figure 6.42, and worse yet, extend beyond the z = 0 plane! The operation of projecting these polygons will cause problems, so that's something else to think about. Of course, if all the polygons are small and their longest edge is smaller than the distance from the near clipping plane and the z = 0 plane, you can simply trivially reject whole polygons that fall into that range and not clip them, and then do pixel image space clipping during rasterization.

Playrix Ex. 1023, page 592

09 0672318350 CH06 05•02•2003 2:40 PM Page 568

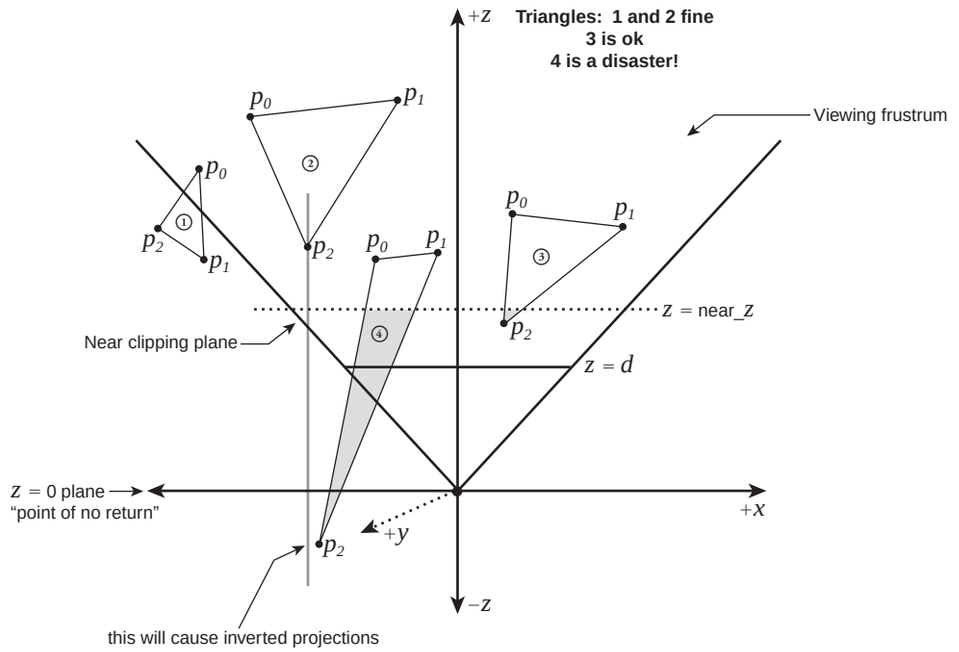


FIGURE 6.42 Triangles piercing the near clipping plane and z = 0 plane.

The moral of the story is that we are going to have to think long and hard about data structures, depending on the types of geometry we will envision and the clipping process we want to use. These factors will play into the representation of polygons, and will be the deciding factor if polygons can have more than three vertices.

You might say, "Let's just have the rule that all models must be built triangles, and then if a polygon ends up with more than three vertices during the 3D pipeline and clipping phase, we can handle those as a special case, either via triangulation, or we can have a special rasterizer for *n*-sided polygons." Again, this is a reasonable approach, and I have used it, too. So in conclusion, you need to think about all these factors at some point, but for now let's just keep things simple.

Defining Polygons

For our first shot at defining polygons, we are going to use the triangle rule and assume that no clipping ever occurs in world space that would necessitate the addition of vertices. That is, if we do any clipping in world space or camera space, we will throw the entire polygon away or keep it and let the rasterizer clip the polygon to the screen during rasterization (this will be fairly easy in our first engine, because it will be wireframe). This simplification enables us to get through this chapter without knowing about 3D clipping

Playrix Ex. 1023, page 593

09 0672318350 CH06 05•02•2003 2:40 PM Page 569

algorithms, which we want to hold off on. Anyway, when we define a polygon, we first have to decide a few things about our 3D worlds, such as what are we going to try and model?

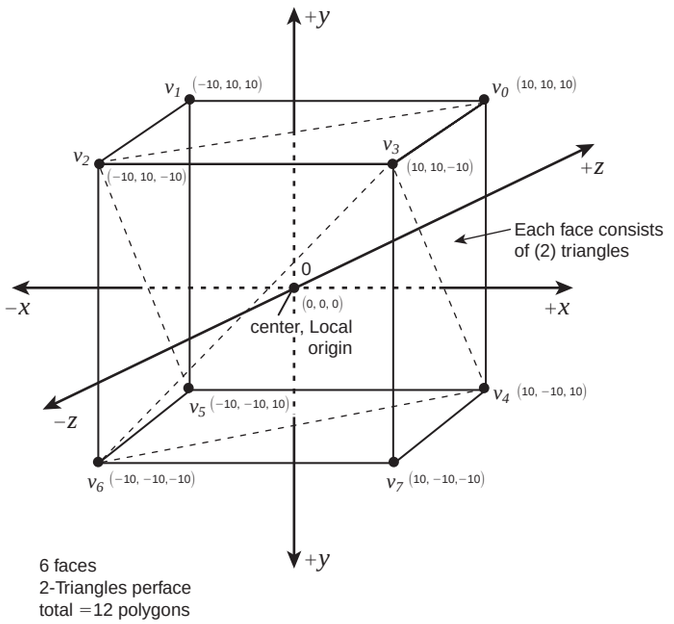


FIGURE 6.43 Modeling a cube.

For example, suppose that you want to model the 3D cube in Figure 6.43. The cube has six faces, which can each be modeled by two triangles (remember, no quadrilaterals allowed). That's a total of 12 polygons and 36 vertices. Or is it? Yes, there are 12 polygons, but are there 36 vertices, or 8 vertices? The answer depends on how you want to store the model. If we decide that every polygon is a unique entity, we would define it as a list of three vertices:

```
struct typedef POLY_EX_TYP_1
{
POINT3D v[3]; // the vertex list
} POLY_EX_1, *POLY_EX_1_PTR;
```

And to define any polygon, we would use something like this:

```
POLY_EX_1 face_1 = { {x0, y0, z0},
                     {x1, y1, z1},
                     {x2, y2, z2}, };
```

Playrix Ex. 1023, page 594

09 0672318350 CH06 05•02•2003 2:40 PM Page 570

That's fine, and works okay. But it's a waste of space, because the cube only has eight total vertices, and if we define a cube with this structure, we will end up with 12 polygons, each containing three vertices. We should be able to reuse the vertex data better. Hence, we can come up with a more indirect method to define a polygon based on a vertex list.

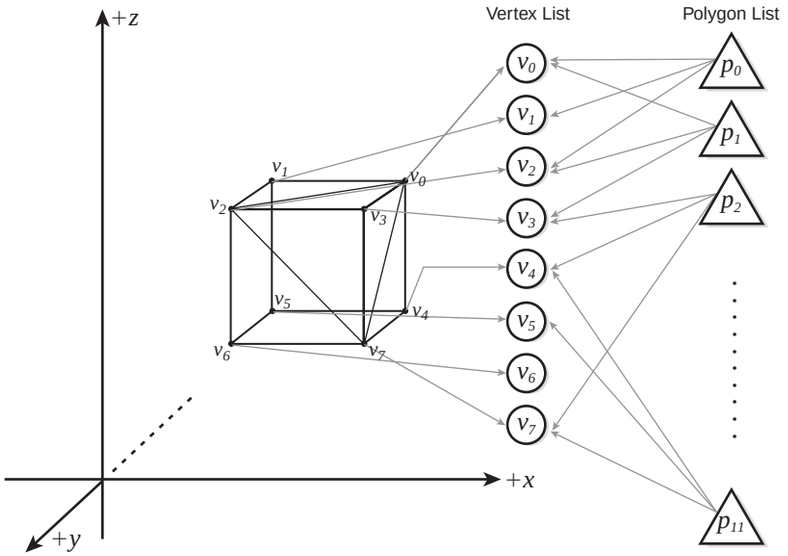


FIGURE 6.44 Geometry can be defined by a vertex list.

A *vertex list* is nothing more than a list of vertices that we reference to construct geometry, as shown in Figure 6.44. Therefore, we first define the vertices of the cube as a list or array, and then define the polygons based on pointers or references to that array. For example, here's how we might define a vertex list:

```
POINT3D vertex_list[NUM_VERTICES]; // the vertex list
```

Then we would initialize each vertex with the proper data from the eight points of the cube:

```
vertex_list[0] = {x0, y0, z0};
vertex_list[1] = {x1, y1, z1};
vertex_list[2] = {x2, y2, z2};
vertex_list[3] = {x3, y3, z3};
vertex_list[4] = {x4, y4, z4};
```

Playrix Ex. 1023, page 595

09 0672318350 CH06 05•02•2003 2:40 PM Page 571


```
vertex_list[5] = {x5, y5, z5};
vertex_list[6] = {x6, y6, z6};
vertex_list[7] = {x7, y7, z7};
```

Now we can define a new polygon type that uses a vertex list to generate the polygon:

```
struct typedef POLY_EX_TYP_2
{
POINT3D_PTR vlist; // the vertex list itself
int vertices[3];   // the indices into the vertex list

} POLY_EX_2, *POLY_EX_2_PTR;
```

Now you can see the indirection. Instead of wasting space in each polygon storing the same vertices over and over, there is a single vertex list referenced by each polygon, so now we can define a single polygon

```
POLY_EX_2 face_1 = {vertex_list, 0, 1, 2};
```

which states that `face_1` is based on the vertex list `vertex_list`, and consists of the vertices (0,1,2), which equal {x0, y0, z0}, {x1, y1, z1}, {x2, y2, z2}, respectively.

There's a another little detail here that's important when we get to solid modeling, and that's *vertex ordering*. That is, when we define polygons or triangles, we need to order them in a specific way, so that we can determine what side is *outward*.

Referring to Figure 6.45, we see a triangle with its vertices ordered in *clockwise* order. Remember when we talked about back-face removal? The only way to perform back-face removal is to first decide what side is the front and back of a polygon, and then project a normal from it (of course, if the polygon is visible from both sides, this is irrelevant). To do this, we need to use a vertex ordering scheme, so we can compute the outward normal in a uniform manner.

If we are using a left-handed system and a clockwise vertex ordering, we can use the *left hand rule* by following the incremental vertex numbers, and our thumbs will point in the outward direction. Or in other words, if we create a vector u = $\mathbf{p}_0$->$\mathbf{p}_1$ and v = $\mathbf{p}_0$->$\mathbf{p}_2$, and then compute the cross product of u × v, this will be the normal n = <v_x, v_y, v_z>, as shown in the figure. This is, of course, the direction of the outward face. This vector n is exactly what we will use during back-face removal (if there is any). Therefore, for this book, we will always use a clockwise vertex ordering that will define the outward direction of any polygon. However, this of course only works for a left-handed system. In a right-handed system, we would have to invert the sign of the normal or use a right-handed rule. It's all in how you want to look at it.

Playrix Ex. 1023, page 596

09 0672318350 CH06 05•02•2003 2:40 PM Page 572

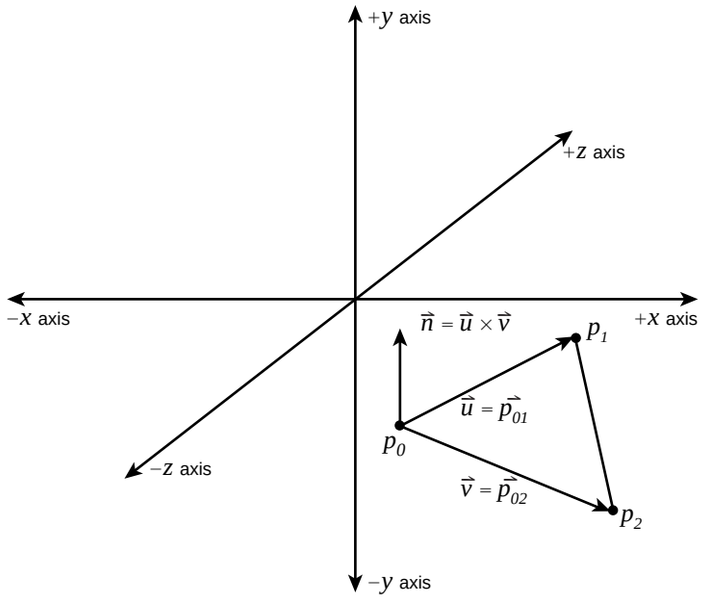


FIGURE 6.45 Vertex ordering.

TIP

Note that for any *n*-sided polygon, there are *n* different orderings that will keep the clockwise ordering (or counterclockwise, for that matter) satisfied—meaning the orderings are not unique. Many times though, if you can order the vertices so that lower numbers are accessed first, you can get better data coherence. For example, suppose that the order (0,5,12) is a clockwise ordering of a polygon. That's a good order to leave it in, even though (12,0,5) and (5,12,0) are also valid because the sequence is the same. But in both cases, data is accessed out of order in the vertex arrays, so the (0,5,12) order will end up having better access characteristics in most cases. However, many times accessing the last element, forcing the cache to be updated, *then* accessing the first elements, is faster.

Anyway, getting back to vertex lists—the vertex list representation is a nice representation, and we get a lot of cool freebies from using it. For example, if we want to transform the object, we don't have to transform the polygons it's made of; we only transform the vertex list. Moreover, when you have vertex copies that represent the same vertex, then many times you get *numerical drift*. That is, you have two polygons that share a common vertex, but each has its own copy of the vertex. Then you transform each polygon and its own copy of the vertex, but during the transform something happens and a slight error occurs in one of the copies, so now the two vertices that are supposed to represent the same vertex don't! This can cause cracking in your geometry. Just take a look at *Tomb Raider* to see what I'm talking about.

Playrix Ex. 1023, page 597

09 0672318350 CH06 05•02•2003 2:40 PM Page 573

Now let's look at the dark side of the coin relating to using vertex lists. Many times as you get farther down the pipeline, you want to separate polygons and vertex lists. The reasons are many, but for example, imagine you have a mesh that has 1,000 vertices, and only one polygon is visible. Because all transformations are done at the vertex list level, you still end up transforming all that data for nothing—just for that single polygon! Of course, there are tricks to get around this, but it's an issue, so keep it in mind. Also, many times it's easier to write a rasterizer that just blasts polygons based on three vertices in a list, and not a list of pointers to a vertex list. Again, this all depends on the final architecture.

Finally, during the clipping stage, it might be very hard to clip the geometry if it's a vertex list and you don't really look at it as polygons. Then again, if you clip one vertex, all polygons that share that vertex will be clipped, but I think you are starting to see that there are a lot of things you have to think about.

With all that in mind, we are going to use a hybridized approach and define and transform all our objects based on vertex lists. However, at some point we will convert them to single polygons, even if they have common vertices. We will change this later, but for now it will simplify things.

Alright—there's yet another detail that I have glossed over that I want to take a look at before we make our final first draft of the polygon structure: the decision to use 3D or 4D homogenous coordinates. This is a tough question, but the right way to do it is to use both. In the end, we will use 3D coordinates and make assumptions that there's a phantom w = 1 with all of them and use the special 4×3 matrix multiplier code that we have written. For now I think we should use 4D points along with 4×4 matrices, so the next couple of chapters will further illustrate the use of matrices to perform clipping, perspective, and projection in a uniform manner. Although we all know that we don't need the extra w = 1 that the 4D homogenous coordinates give us and that we can assume it with special code, I don't want to make that assumption for now. Hence, we will assume that all points and vertices are in 4D homogenous coordinates with w = 1 in most cases, just to make the math and matrix operations uniform. With that in mind, let's create a more solid vertex list and polygon data structure based on these conventions.

Our First Polygon

We have left out a lot of details about the polygons, because at this point, we don't know enough about rendering to add them. However, for the next chapter, where we design a working wireframe engine, we will need at least the following for each polygon:

- Color—General color of the polygon. This could be a color index for 8-bit color, or an actual 16-bit or 24-bit RGB value for high-color modes. It could also be something else that you make up.
- State—State of the polygon. This will hold state information that changes dynamically as the polygon is processed. For example, is the polygon active, is it clipped, is it damaged?—things like that.

Playrix Ex. 1023, page 598

09 0672318350 CH06 05•02•2003 2:40 PM Page 574

- Attributes—Physical attributes of the polygon. This might hold a number of flags that describe the polygon. For example, is it two-sided, does it reflect light, is it made of steel, is it transparent, and so forth.

As stated in our conventions, we are always going to assume that all polygons never have more than three vertices—just for now. With that in mind, here's the first version of the polygon structure:

```
typedef struct POLY4DV1_TYP
{
int state;    // state information
int attr;     // physical attributes of polygon
int color;    // color of polygon

POINT4D_PTR vlist; // the vertex list itself
int vert[3];       // the indices into the vertex list

} POLY4DV1, *POLY4DV1_PTR;
```

NOTE

The naming convention is [*root*][*coordinate type*][*version*]. `POLY` is the type of this object, `4D` is the coordinate type, and `V1` is the version.

Of course, the vertex list that `vlist` points to is of the type `POINT4D[]`. We're not going to further define `state`, `attr`, or `color`, because we want to keep them abstract and don't want to lock ourselves down. Let's continue on with another data structure that we might need, which I'm going to call a *face*.

Remember that we might need a simple list of polygons not associated with objects or general vertex lists during the rasterization phase. That is, we want a simple array or linked list of polygon faces that are each self-contained that we can send to the rasterizer. We might want to have this capability, so let's just create a data structure right now called `POLYF4DV1` to hold this object. It should be identical to a `POLY4DV1` structure, except that it's self-contained and doesn't reference an external vertex list.

```
typedef struct POLYF4DV1_TYP
{
int state;    // state information
int attr;     // physical attributes of polygon
int color;    // color of polygon

POINT4D vlist[3];  // the vertices of this triangle
POINT4D tvlist[3]; // the vertices after transformations
```

Playrix Ex. 1023, page 599

09 0672318350 CH06 05•02•2003 2:40 PM Page 575


```
POLYF4DV1_TYP *next; // pointer to next polygon in list
POLYF4DV1_TYP *prev; // pointer to previous polygon in list

} POLYF4DV1, *POLYF4DV1_PTR;
```

Notice that there's a `next` and `prev` pointer. These are to help form a linked list of polygons like the one shown in Figure 6.46 that we can pass the rasterizer—we may or may not need them, but just some forward thinking. Also, notice that there are two copies of vertices `vlist[]` and `tvlist[]`. The second list `tvlist[]` is used to hold transformed versions of the first list. That is storage, so we don't have to overwrite the polygon as we manipulate it—again, we may or may not need this. Later, we might have extra space for world coordinates, camera coordinates, perspective coordinates, and screen coordinates!

The Poly F4DV1 Structure

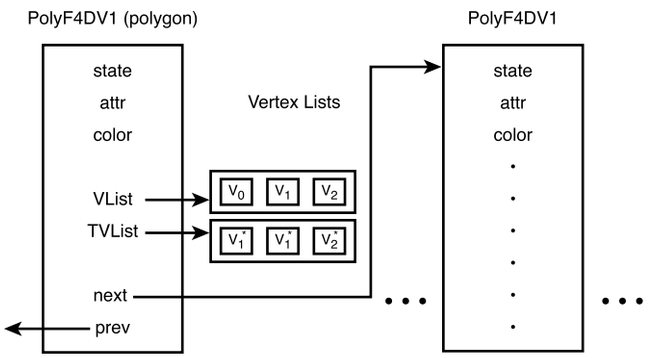


FIGURE 6.46 The `POLYF4DV1` structure.

NOTE

Make a note that we cannot destroy the original 3D models' local coordinates. We must either copy them or transform them into another data structure at some point, because they are always the source of everything. On the other hand, if you perform local transformations like rotation and scaling, you usually want the model to stay that way, so it's okay to overwrite the original vertex data.

Alright, we have a vertex list, a polygon structure, and a face structure that we can use for rendering, so we have just barely enough to work with. Now let's increase the level of hierarchy a bit and talk about objects.

Playrix Ex. 1023, page 600

09 0672318350 CH06 05•02•2003 2:40 PM Page 576

Defining Objects

An object in a 3D game is usually a collection of polygons that defines a single object, as shown in Figure 6.47. However, in a more advanced 3D system, an object might be a collection of objects interconnected with links, as shown in Figure 6.48. We will deal with the second type later, but for now, let's deal with the simple object that is composed of a rigid mesh of polygons with no moving parts, as shown in Figure 6.47. In this case, we have an object that is composed of a set of polygons that is itself based on a vertex list. With this abstract model, we can very easily come up with a data structure to hold a general object. However, it's at this point that we have to make some decisions about data storage in relation to the 3D pipeline.

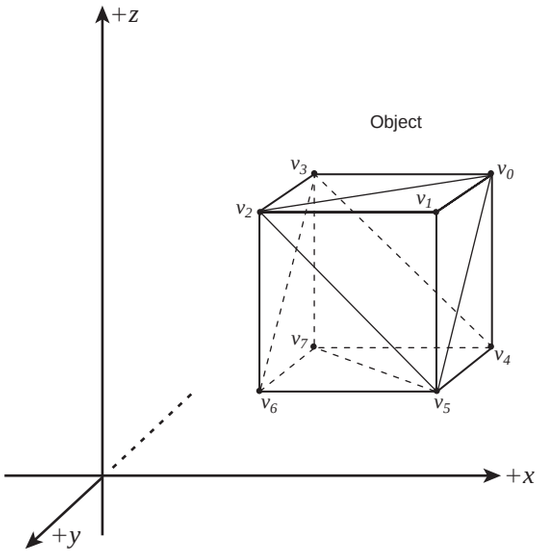


FIGURE 6.47 A single object made of polygons.

Remember that there are local coordinates, world coordinates, camera coordinates, perspective coordinates, and screen coordinates. The question is, do we need storage for each set of vertices as they propagate down the pipeline? The answer depends on a lot of things, but at the very least, we need the local coordinates of every object. Then the determination of necessary secondary storage really depends on how the 3D pipeline works internally. This means that if we pass a single object to the pipeline, do we put the responsibility on the pipeline or on the object for secondary storage.

This is a hard question, and there's no good answer. I have done it both ways—that is, create objects that internally have storage for local, world, and camera coordinates. On the other hand, I have also completely implemented it in the viewing pipeline.

Playrix Ex. 1023, page 601

09 0672318350 CH06 05•02•2003 2:40 PM Page 577

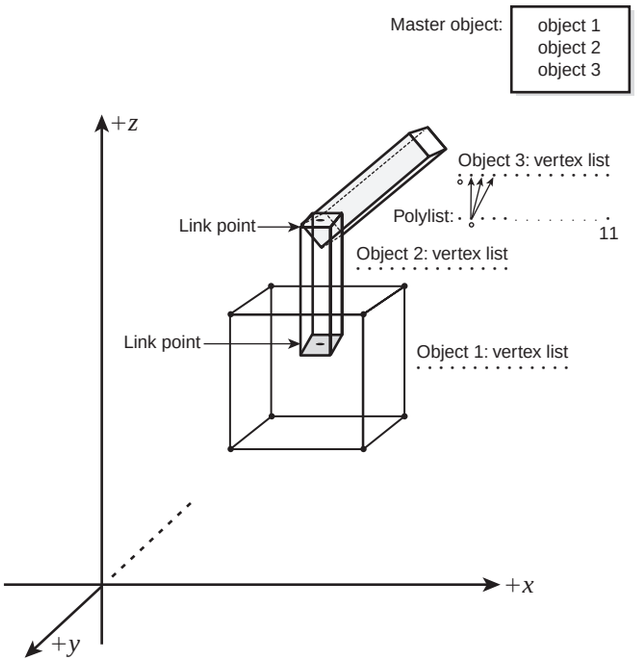


FIGURE 6.48 An interlinked object composed of many sub-objects.

For now, I think that I want to keep things relatively simple, so we are going to think of an object as a container of everything—even the transformations of the pipeline. However, we are only going to have storage for the local coordinates, along with the transformed coordinates. The transformed coordinates could be world, camera, perspective, or screen; we only know which by finding at which stage the data is in the pipeline. This minimizes storage. Okay, so that takes care of that. Now, what else do we need to define an object? Well, we need the number of vertices, maybe some attributes, state, and a name or `id`, along with a bounding radius that we will use later. So, our first shot at an object will look like this:

```
// an object based on a vertex list and list of polygons
typedef struct OBJECT4DV1_TYP
{
int  id;           // numeric id of this object
char name[64];     // ASCII name of object just for kicks
int  state;        // state of object
int  attr;         // attributes of object
```

Playrix Ex. 1023, page 602

09 0672318350 CH06 05•02•2003 2:40 PM Page 578


```
float avg_radius;  // average radius of object used for collision detection
float max_radius;  // maximum radius of object

POINT4D world_pos;  // position of object in world

VECTOR4D dir;       // rotation angles of object in local
                    // cords or unit direction vector user defined???

VECTOR4D ux,uy,uz;  // local axes to track full orientation
                    // this is updated automatically during
                    // rotation calls

int num_vertices;   // number of vertices of this object

POINT4D vlist_local[64]; // array of local vertices
POINT4D vlist_trans[64]; // array of transformed vertices

int num_polys;        // number of polygons in object mesh
POLY4DV1 plist[128];  // array of polygons

} OBJECT4DV1, *OBJECT4DV1_PTR;
```

The first thing that you will notice is the incredible waste of space that `OBJECT4DV1` exhibits because of the static arrays. This is not an oversight, but on purpose, because I want to avoid memory allocation/deallocation issues at the juncture. Obviously, later versions will dynamically allocate the vertex and polygon lists. But for now, 64 vertices and 128 polygons should be more than enough to model cubes, rocks, and sex toys? Also notice that there are fields for the average and maximum radius of the object—these are to help with object culling.

And of course, there's a general position of the object `world_pos`, along with a general orientation of the object in its local space, named `dir`. You see, when we transform the object's orientation in local coordinates, we will modify the local coordinates and lose them, so we need some method to track these changes. Of course, we could use the convention that each time an object is processed through the pipeline, we perform all the local rotations to it on the fly and leave the original unscathed, but this might be a waste of time if we have 1,000 rocks in a desert that never rotate. In many cases, 3D engines will rotate objects in their local spaces and overwrite the local coordinates, but this rotation needs to be tracked, and that's what `dir` is for.

Moreover, we can store the angles there, or even a unit vector that's aligned with the z-axis initially (dir = <0,0,1>) and rotate each time along with the object, so we can keep track of the object's orientation relative to load time. Additionally, the basis vectors u_x, u_y, u_z are also to keep track of the object's orientation. That is, every time the object is transformed, so are u_x, u_y, u_z, which tracks the orientation of the object. For example, suppose

Playrix Ex. 1023, page 603

09 0672318350 CH06 05•02•2003 2:40 PM Page 579

we want to define a basic cube (as you have seen about ten times now), based on the following vertex list:

p0(10,10,10)
p3(-10,10,10)
p2(-10,10,-10)
p1(10,10,-10)
p4(10,-10,10)
p5(-10,-10,10)
p6(-10,-10,-10)
p7(10,-10,-10)

Here's the code that does it:

```
// define the cube object manually (very tedious)
OBJECT3DV1 cube_1;

cube_1.id    = 0;         // unused for now
cube_1.state = 0;         // unused for now
cube_1.attr  = 0;         // unused for now

cube_1.avg_radius  = 17.3;  // average radius of object
cube_1.max_radius  = 17.3;  // maximum radius of object
                            // sqrt(10*10+10*10+10*10)


cube_1.world_pos.x = 0; // position of object in world
cube_1.world_pos.y = 0; // who cares right now
cube_1.world_pos.z = 0;

cube_1.local_ang.x = 0; // the current orientation of
cube_1.local_ang.y = 0; // the object since loading
cube_1.local_ang.z = 1; // this will be default for
                        // all objects

cube_1.num_vertices =  8; // number of vertices

// create a vertex list to simplify initialization
POINT3D temp_verts[8] = { {10,10,10},     // p0
                          {10,10,-10},    // p1
                          {-10,10,-10},   // p2
                          {-10,10,10},    // p3
                          {10,-10,10},    // p4
                          {-10,-10,10},   // p5
                          {-10,-10,-10},  // p6
                          {10,-10,-10}};  // p7
```


Playrix Ex. 1023, page 604

09 0672318350 CH06 05•02•2003 2:40 PM Page 580


```
// define the local vertices

for (int vertex=0; vertex < 8; vertex++)
    {
    cube_1.vlist_local[vertex].x = temp_verts[vertex].x;
    cube_1.vlist_local[vertex].y = temp_verts[vertex].y;
    cube_1.vlist_local[vertex].z = temp_verts[vertex].z;
    cube_1.vlist_local[vertex].w = 1;
    } // end for index

cube_1.num_polys = 12; // number of polygons

// define the polygons now, note that I am setting
// the state, attr, and color all to 0 for this exercise
// and the vertex pointer is pointing to the local coords
// but it doesn't really matter what list we point it to
// since we know that all polygon defined in this object
// are based on ITS vertex list and not something external

// create a list of triads that define each polygon to
// simplify initialization, note all in clockwise
// order, so we have an outward direction
int temp_poly_indices[12*3] = {
0,1,2, 0,2,3,   // polygons 0 and 1
0,7,1, 0,4,7,   // polygons 2 and 3
1,7,6, 1,6,2,   // polygons 4 and 5
2,6,5, 2,3,5,   // polygons 6 and 7
0,5,4, 0,3,5,   // polygons 8 and 9
5,6,7, 4,5,7 }; // polygons 10 and 11

// now initialize each triangle
// triangle 0

for (int tri=0; tri < 12; tri++)
    {
    // all of these are unused as of yet
    cube_1.plist[tri].state = 0;
    cube_1.plist[tri].attr  = 0;
    cube_1.plist[tri].color = 0;

    // point the vertex list to local coords
    cube_1.plist[tri].vlist = cube_1.vlist_local;

    // now assign the vertex indices
    cube_1.plist[tri].vert[0] = temp_poly_indices[tri*3+0];
```

Playrix Ex. 1023, page 605

09 0672318350 CH06 05•02•2003 2:40 PM Page 581


```
cube_1.plist[tri].vert[1] = temp_poly_indices[tri*3+1];
cube_1.plist[tri].vert[2] = temp_poly_indices[tri*3+2];
} // end for
```

Wow, that seems like a ton of code for something so simple as a cube—and it is! This is one of the first problems with 3D graphics: the huge amounts of data. You would never represent a bitmap as individual bytes of data in a game, and likewise you would never represent an object or mesh as data in a real 3D game. Meshes and worlds are always loaded in from files because they can be made of hundreds, thousands, or millions of vertices. For now, we are going to use manual input along with some primitive ASCII-based file formats to represent our 3D objects, until we get more sophisticated.

At this point we have the cube, and we would proceed to push it through the 3D pipeline with a camera, viewing frustrum, and all the stuff we have been talking about. Then at the end, we would render the object, maybe with shaded polygons or wireframe, but that's for the next chapter.

Representing Worlds

Right now, our view of a 3D world is a composition of 3D objects, in which each object is a collection of polygons that are based on a vertex list (or maybe not). This is fine, but what about if we want to make a game like *Quake*, or a tank simulation with a huge amount of terrain? Sure, the objects in the game map nicely to our objects, but the indoor environment of *Quake* or the huge terrain maps of an outdoor tank simulation might not fit into our little object definition.

Indoor and outdoor worlds (or more generally, environments) are usually special cases of objects, and more advanced data structures must be employed to represent them properly because of their sheer size. For example, a typical game object of a tank or character might have about 20–500 polygons. This is manageable with the object data structure that we have designed. On the other hand, imagine a world that has tens of thousands, or even tens of millions of polygons. We can't just have a huge list of these polygons and throw them at the pipeline. We need to be smarter about how we load the data in, because at any time only a small fraction of the polygons in a world will be visible. Along those lines, we've already talked about how we might represent a terrain world in some of the previous sections of this chapter, and one of our solutions was to arrange the world as a matrix of sectors. Perhaps the sectors could be objects? On the other hand, some of our algorithms fall apart (object-removal) because there's no sense of objects when dealing with terrain or an indoor environment. On the other hand, these concepts are replaced with the concepts of rooms, areas, and portals, and we use analogous methods to cull them.

Additionally, most of the time terrain and indoor environments don't change much, so we can do lots of precalculations and optimizations on them and employ *binary space partitions*, *octrees*, or *portals* to simplify and minimize the amount of data that we have to process each frame. For instance, an environment can be defined completely in world coordinates because it won't move, thus saving us the local-to-world transformation.

Playrix Ex. 1023, page 606

09 0672318350 CH06 05•02•2003 2:40 PM Page 582

Therefore, in most cases after we decide on the parts of the environment that are visible, we need only push them through the camera and projection matrices.

We will talk about all these concepts as the book progresses, but I want you to fully understand that we can get away with simple wireframe and flat-shaded 3D games with what we have so far. For more advanced games, we are going to have to think of much more robust data structures and algorithms to cut down all the data processed with each frame.

With all that in mind, there's no reason to even try to come up with a 3D data structure right now for general worlds, because we don't know enough about 3D, and truthfully, we need to tailor the design to the world itself. However, we can imagine that for a game similar to *Quake*, we might design the engine so that each room is a separate node in a tree. For every room, there are links to all the other rooms that are visible from that room. Thus, at any time we could refer to this data structure, and based on our position, cull out all but a few rooms. This is shown in Figure 6.49.

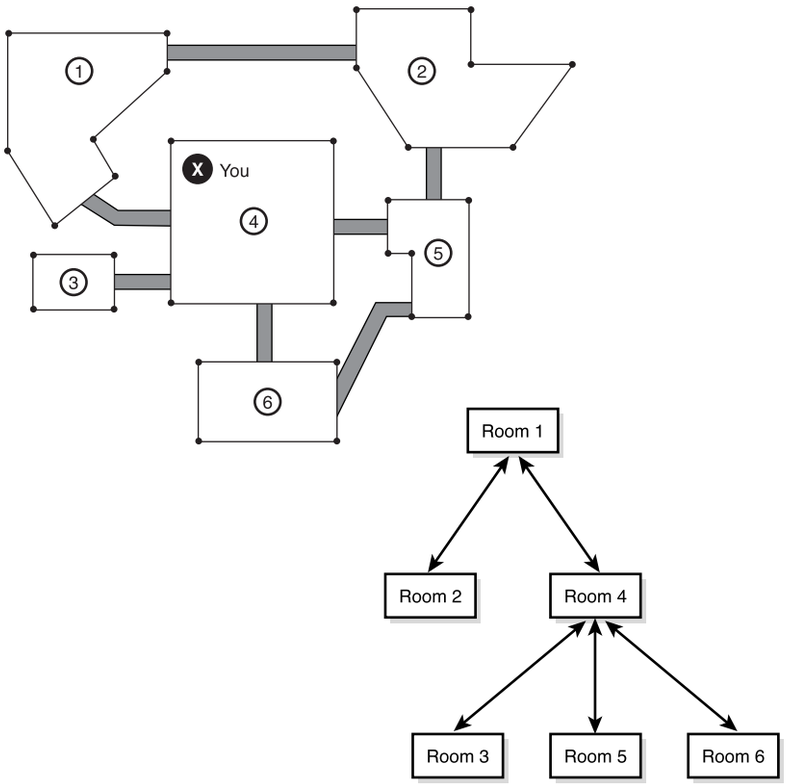


FIGURE 6.49 A possible level data structure.

Playrix Ex. 1023, page 607

09 0672318350 CH06 05•02•2003 2:40 PM Page 583

Of course, this falls apart if we can see outside, or there are lots of windows, and so on. But if you write a game with rooms all connected with long hallways, you can have a billion rooms and the engine would be able to instantly cull them. These are some of the ideas behind BSP trees, octrees, and portals—more on this later, though. I'm getting ahead of myself, or at least the entity that is currently inhabiting my body is …

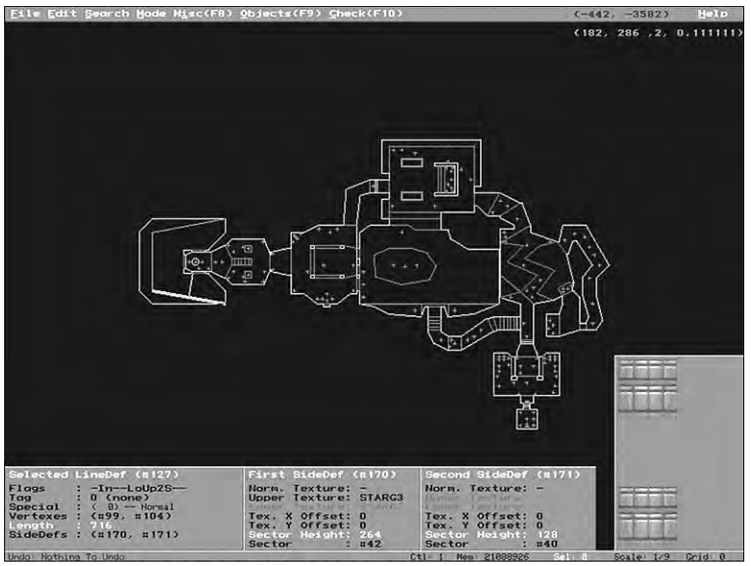


FIGURE 6.50 A screenshot of DoomEdit in action.

3D Tools

Alrighty, then. Where to even begin? Well, making a 3D game these days is a lot of work, mostly because of the 3D aspect of game and the amount of data needed. Alas, one of the most vexing problems facing a game programmer/designer is how on Earth all the data for the game will be generated and worked with. There are many solutions to this problem. For simple 3D games, you might create tools yourself for 3D modeling and world generation. For example, Figure 6.50 shows an old screenshot of the *Doom* editor. As you can see, the tool is bitmapped/line-drawn-based and features a top-down view. The tool reads in various data files and then enables the level designer to place objects, geometry, switches, creatures, lights, and so forth to create the world. DoomEdit and other "build engines" were used to generate the entire 3D world for the games, and were all-in-one solutions. Of course, other game assets like scripts, bitmaps, and sounds were created in other tools.

Playrix Ex. 1023, page 608

09 0672318350 CH06 05•02•2003 2:40 PM Page 584

These days things have become much more complex, because the worlds are totally 3D, with arbitrary geometry. With this in mind, there are two routes to take: You can still write your own 3D tool, but realize that you're going to end up writing a 3D modeling package in essence; or you can use an existing 3D modeler like Caligari trueSpace, Maya, LightWave, 3D Studio Max, Blender, or Moray (coupled with the Persistence of Vision POV ray-tracer), completely design your objects and worlds with these modelers, and then interface the tools to your game engine via extra software and or plug-ins. This is probably the best approach, because you leverage someone else's work. On the other hand, 3D modelers aren't good at placing monsters, adding switches to doors, and so on, but they are getting better, and many of the aforementioned modelers have lots of hooks and API additions that enable you to modify the base functionality of the modeler to your needs.

Originally for this book I was going to use the *Quake* format for 3D worlds, and use something like WorldCraft to create the worlds. However, the developers of WorldCraft have kind of gone out of business (transferred the program to Valve, Inc. now under the name of the Hammer Editor), *Quake* is a proprietary format, and on second thought, I don't want to lock myself into it. Thus, we are going to use one or more of the standard file formats, like Caligari .COB, 3D Studio Max .3DS or .MAX files, .DXF, .NFF (neutral file format) and so on in this book, so you can use commonly available tools and well-documented formats. Moreover, this book is about 3D graphics and game programming, not really writing level design tools, so I want to stick to the point. With that in mind, you can use whatever 3D modeling tool you like, as long as it supports the file format(s) that I am going to read in this book.

Along those lines, I have placed a number of 3D modeler demos on the CD, along with some shareware modelers that are pretty good, too—Moray, for example, rocks. Even if the shareware modelers don't support the file format you like, I have also included a number of 3D file format converters on the CD.

Animation and Motion Data

Additionally, at some point you're going to want to create animated articulated creatures. This gets really complex. If you recall, we learned that you can animate creatures by mesh animations or by animated skeletal animation. In the first case, you'll need to get the 3D modeler to spit out the character mesh key frames and then you will need to read that out. In the second case, you'll need to be able to read in a mesh with articulation points and then pump motion data (like BioVision motion files) through the mesh links to animate the creature or object (if we go that far). This is again a very complex task, and the tools will get you the object and even give you the motion data (or you can make it or buy it), but you have to write the code that feeds the data into your engine. We will talk more about this later. Tools like 3D Studio Max, Maya, and trueSpace all have good character animation packages that will let you do both. However, in the end we may use a more basic keyframe system like *Quake*'s .MD2 format — we will see.

Playrix Ex. 1023, page 609

09 0672318350 CH06 05•02•2003 2:40 PM Page 585

Loading Data from the Outside World

As I said, we are going to write code as we need it, but just to get you in the spirit of things, I want to show you a couple of file formats that we can use and just let you see what they look like. The whole point of a file format is to be able to represent 3D objects with enough information for our engine. At this point, we don't need to know anything about textures, lighting, and so on—we just basically want a description of an object as a vertex list and a polygon mesh.

However, as the book progresses, we are going to want to know a lot! For example, if we use the 3D modeler to create a tank, for example, we are going to want to know the vertex list, polygon mesh, texture coordinates, polygon attributes, and so on, so a more sophisticated file format will be needed. Additionally, if we want to model entire worlds with a 3D modeler, the modeler will have to support exported light sources or tagged information that our engine can use, and so on.

Describing all the file formats in complete detail is beyond the scope of this book, and beyond the scope of this chapter for that matter, but I just want to give you a taste for some of the more popular formats.

PLG Files

The PLG file format was invented by the people who made REND386, one of whom is Bernie Roehl (actually, I think he wrote the entire thing). The format is simple and to the point, so it's a good place to start. Unfortunately it lacks a bit in the texturing and two-sided polygon area, but it will suffice with a bit of finesse as a quick way for us to hand-edit models if we have to. The main reason to use it is that there are 3D modelers that generate PLG files, and utilities that convert .DXF files to PLG files. But most of all, writing a parser to read the files only takes an hour or two instead of a week, like it would for one of the more advanced formats.

PLG Header Line

A PLG file starts off with the name of the object and the number of vertices and polygon faces in the format:

```
object_name num_vertices num_polygons
```

For example,

```
pyramid 5 5
```

which says that the data to follow is for the object `pyramid`, which has five vertices and five faces.

Playrix Ex. 1023, page 610

09 0672318350 CH06 05•02•2003 2:40 PM Page 586

PLG Vertex List

The next part of the data are the vertices in order of 0 to *n*. Each vertex should be separated by whitespace and be in x-y-z order. For the pyramid shown in Figure 6.51, the vertex data is

```
0   20  0
 30 -10  30
 30 -10 -30
-30 -10 -30
-30 -10  30
```

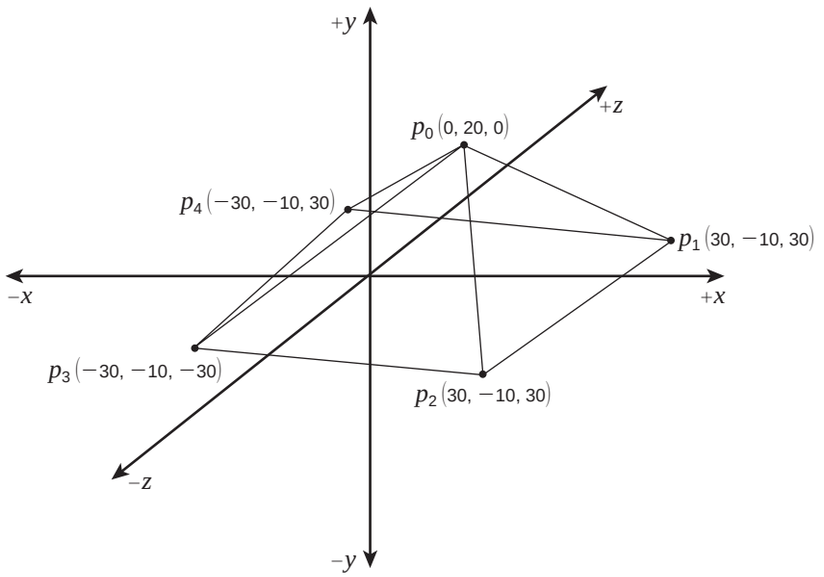


FIGURE 6.51 Five-faced pyramid.

PLG Polygon List

The next part of the file is the polygon description which is a series of lines that describe the color of the polygon, the number of vertices in the polygon, and the index number of the vertices. The format is

```
surface_description n v1 v2 v3 … vn
```

where *surface_description* is a surface descriptor we will get to in a moment, *n* is the number of vertices, and *v1*, *v2*, ... *vn* is the vertex list in counter-clockwise order. Note that in the actual file, there are no commas. In our pyramid example, the polygon list would look like this:

Playrix Ex. 1023, page 611

09 0672318350 CH06 05•02•2003 2:40 PM Page 587


```
10  3 0 2 1
10  3 0 3 2
10  3 0 4 3
10  3 0 1 4
10  4 1 2 3 4
```

which means that there are five polygon faces. The first number on each row is called the surface descriptor and the next number is the number of vertices in that particular polygon. For example, the polygon in the second row has three vertices (0,3,2) and the polygon in the fifth row has four vertices (1,2,3,4). Remember, we are going to force all our models to be triangulated, but this is for illustrative purposes, to show that the format supports *n*-sided polys.

The only mystery at this point is the first number in each row, which is called the *surface descriptor* (in the case of the pyramid, this is 10 for all faces). Let's take a look at the meaning of the surface color descriptor.

PLG Surface Descriptor

The first number in each of the polygon definitions is called the *surface descriptor*. The surface descriptor can either be a decimal integer or in hex, with a 0x (or 0X) followed by a hexadecimal integer value. The surface descriptor is a 16-bit value, which is interpreted as bit fields in the following manner:

$H_{15}\ R_{14}\ S_{13}S_{12}\ C_{11}C_{10}C_9C_8\ B_7B_6B_5B_4B_3B_2B_1B_0$

- The R bit is reserved, and should be set to zero.
- If the H bit is set, it indicates that this is a "mapped" surface descriptor, and the lower 14 bits are taken to be an index into a surface map.
- If the H bit is zero, the SS bits are interpreted as shown in Table 6.1 following:

TABLE 6.1 Bit Definition for SS Bits of PLG Surface Descriptor

$S_{13}S_{12}$	Definition
00	This polygon is *solid shaded*; that is, it should be drawn in a fixed color with no special effects. If the CCCC bits are zero, the BBBBBBBB bits directly specify one of the 256 available colors in 256 color modes. If the CCCC bits are nonzero, they specify one of sixteen hues, and the top four bits of BBBBBBBB specify which shade of that hue to use.
01	This polygon is *flat shaded*; that is, it should be drawn with a constant shading that is determined by the angle at which light is striking it. Thus, as the polygon moves around, its apparent brightness will change. The CCCC bits specify one of sixteen hues, and the bottom 8 bits BBBBBBBB represent the "brightness" of the color. This brightness value is multiplied by the cosine of the angle between the polygon's normal vector and the vector from the polygon to the light source. The result is used to specify an offset into the given color's array of shades. Note that if the CCCC value is 0, the color will always be black.

Playrix Ex. 1023, page 612

09 0672318350 CH06 05•02•2003 2:40 PM Page 588

TABLE 6.1 Continued

$S_{13}S_{12}$	Definition
10	This polygon should be treated as being "metallic." The CCCC bits (which should be nonzero) specify one of the 16 hues, and the top five bits of the BBBBBBBB value are used as an offset into a range of shades to cycle through to give the metallic effect; that is, a starting offset into the color cycle.
11	This polygon should be treated as being "transparent." It is just like surface type 10, except that alternating rows of dots are used instead of solid colors, allowing you to "see through" the polygon.

Note that Table 6.1 reflects the original definition of the PLG file format, and is a bit antiquated because it's obviously designed for 4- and 8-bit color modes. Although the 8-bit support will work for us here and there, we really need 16-bit colors. So what we will do is slightly modify the format when we get to using it in the next chapter to support 16-bit RGB colors. Of course, programs that export the files won't know this, but at least we will get hand-made support for our experiments.

Loading Objects with PLG Files

Loading in a PLG file isn't too much trouble. Basically, PLGs are standard ASCII files with one or more lines of text that describe the object. The only thing that makes reading PLG files a bit complicated is that color descriptors can be in either hex or decimal format, depending on a leading `0x`, and the files can have comments within them that can be started with the character `#`. Anyway, at this point we aren't going to write a parser, but it doesn't look that hard to do: not much more than reading lines one at a time and parsing out some numbers. Relatively speaking, this is a great format to hand-generate models for experimental uses, and we will probably use this in the next chapter just to get going. Here's the final object though, so you can see it in its entirety:

```
pyramid 5 5

# vertex list

 0   20  0
 30 -10  30
 30 -10 -30
-30 -10 -30
-30 -10  30

# polygon list

10  3 0 2 1
10  3 0 3 2
```

Playrix Ex. 1023, page 613

09 0672318350 CH06 05•02•2003 2:40 PM Page 589


```
10  3 0 4 3
10  3 0 1 4
10  4 1 2 3 4
```

That's basically the entire PLG format description, so you can see why I like it. It's simple, down and dirty, gives you something to start with, and it's easy to write a reader for it. Of course, most modelers don't support it directly because it's so basic, but there are converters. The only issue is that PLG files refer to a right-handed system with vertices in counterclockwise order, so we must take that into consideration when loading objects exported from a tool, because we are using a left-handed system.

NFF Files

The next file format that is fairly easy to work with is called *neutral file format (NFF)*. I first used this format many years ago on VR games and it seemed to work well, so I think we might support it. It was originally developed by Eric Haines of Sense-8. Those were the days … Anyway, the NFF file format is about up to version 3.1, so we will cover that.

The NFF format was designed to define simple worlds in addition to objects, in contrast to the PLG format, which just defines objects. Entities like the camera, background color, positions of lights, and so forth are supported in the NFF format. These features have very minimal impact on the parsing complexity of NFF, so it's a great format for what we want to do, as well as testing algorithms and try things.

3D Entities Supported by NFF

NFF supports the following entities:

- A simple perspective frustum
- A background color description
- A positional (versus directional) light source description
- A surface properties description
- Polygon, polygonal patch, cylinder/cone, and sphere descriptions

NFF files are output as lines of text. For each entity, the first field defines its type. The remainder of the line and possibly other lines contain further information about the entity itself. Entities types include the following:

`v`—Viewing vectors and angles

`b`—Background color

`l`—Positional light location

`f`—Object material properties

`c`—Cone or cylinder primitive

Playrix Ex. 1023, page 614

09 0672318350 CH06 05•02•2003 2:40 PM Page 590

s—Sphere primitive

p—Polygon primitive

pp—Polygonal patch primitive

Let's take a brief look at the more important entities that we will deal with. I'll bold all the keywords and leave the parameters alone, which are usually floating point values.

Viewpoint Location Format The viewpoint location defines the viewpoint, along with some of the viewing frustrum parameters we have covered, but not all. Here it is:

```
v
from Fx Fy Fz
at Ax Ay Az
up Ux Uy Uz
angle angle
hither hither
resolution xres yres
```

Parameters:

from—The viewpoint location in world coordinates

at—A "look at" position to be at the center of the image, in world coordinates

up—A vector defining which direction is up relative to the at vector

Angle—Viewing angle in degrees, defined from the center of the top pixel row to the bottom pixel row, and left column to right column.

hither—The distance to hither/near clipping plane (if any) from the viewpoint/eye.

resolution—The resolution of the view plane in pixels.

NOTE

Note that no assumptions are made about normalizing the data (for example, the from-at distance does not have to be 1). Also, vectors are not required to be perpendicular to each other. Also, the far clipping plane is at infinity by default, and the aspect ratio is 1.0.

By convention, a view entity must be defined before any objects are defined, so if the NFF file is used to define the view point, the engine knows where to view from!

Background Color Format The background color is simply an RGB with values, with each channel from 0 to 1:

```
b R G B
```

Playrix Ex. 1023, page 615

09 0672318350 CH06 05•02•2003 2:40 PM Page 591

If no background color is set, assume RGB = {0,0,0}.

Positional Light Format Lights are defined by their position and color as follows:

```
l X Y Z [R G B]
```

where `X Y Z` is the position of the light, and `R G B` is its color (which is optional). Also, all light entities must be defined before any objects are defined. However, lights that aren't colored will be assigned a nonzero intensity, which is undefined.

Fill Color and Shading Parameters Format The fill color and shading specifies the lighting properties of all entities that follow until a new fill color and shading are specified.

```
f red green blue Kd Ks Shine T index_of_refraction
```

Parameters:

`red`, `green`, `blue`—RGB values from 0.0 to 1.0

`Kd`—The diffuse reflection component

`Ks`—The specular reflection component

`Shine`—The Phong cosine exponentiation power for highlights

`T`—The transmittance

Most of these parameters don't mean much to you yet because we haven't covered lighting, but in general there's a simple formula that defines the reflection of light off a surface in terms of its diffuse and specular (highlights) values, and that's what most of the preceding parameters are referring to.

Polygon Format A polygon is defined by a set of vertices that must be coplanar. In NFF files, all polygons are one-sided by definition, with the order of the vertices counter-clockwise as you face the polygon in a right-handed system. Also, the first two edges must form a nonzero convex angle, so that the normal and side visibility can be determined by using just the first three vertices.

CAUTION

The NFF file format is based on a right-handed coordinate system. Watch out!

```
p num_vertices
v1.x v1.y v1.z
v2.x v2.y v2.z
.
.
vn.x vn.y vn.z
```

Playrix Ex. 1023, page 616

09 0672318350 CH06 05•02•2003 2:40 PM Page 592

where *num_vertices* is simply the number of vertices in the polygon, and v1 represents each vertex.

Polygonal Patch Format A polygonal patch is defined by a set of vertices and their normals. It's almost identical to a polygon p, but with the added feature that each vertex has a normal along with it.

```
pp num_vertices
v1.x v1.y v1.z n1.x n1.y n1.z
v2.x v2.y v2.z n2.x n2.y n2.z
.
.
vn.x vn.y vn.z nn.x nn.y nn.z
```

where *num_vertices* defines the number of vertices in the polygon patch, and v1 and n1 represent each vertex and its normal, respectively.

> **NOTE**
>
> The NFF format has a number of other features, such as allowing primitives like spheres and cones to be defined, but for the most part, the preceding entities cover everything we will need, except maybe the comment character #.

A Sample NFF File

To finish up our introductory discussion of NFF files, here's an actual 2×2×2 cube in NFF format:

```
v
from 0 2.0 2.0
at 0 0 0
up 0 1 0
angle 40
hither 1
resolution 512 512
l 0 5 0
l 0 2.0 2.0
f 0.000000 0.000000 0.000000 1 0 0 0 0
p 3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
p 3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
```

Playrix Ex. 1023, page 617

09 0672318350 CH06 05•02•2003 2:40 PM Page 593


```
1.000000 -1.000000 1.000000
p 3
-1.000000 1.000000 -1.000000
-1.000000 -1.000000 -1.000000
1.000000 -1.000000 -1.000000
p 3
-1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 -1.000000
p 3
1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 -1.000000 1.000000
p 3
1.000000 1.000000 -1.000000
1.000000 -1.000000 1.000000
1.000000 1.000000 1.000000
p 3
1.000000 1.000000 1.000000
1.000000 -1.000000 1.000000
-1.000000 -1.000000 1.000000
p 3
1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000
-1.000000 1.000000 1.000000
p 3
-1.000000 1.000000 -1.000000
1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
p 3
-1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
p 3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 -1.000000
-1.000000 1.000000 1.000000
p 3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000
```


Playrix Ex. 1023, page 618

09 0672318350 CH06 05•02•2003 2:40 PM Page 594

3D Studio Files

3D Studio and its newer offspring, 3D Studio Max (or just plain Max) are some of the most widely used 3D modelers in the world for both games and the movie industry. The original 3D Studio has been around for a long time, but it was actually an offspring of an even older program called AutoCad, written by AutoDesk. It was AutoCad that originally supported the DXF file format, and then later, when 3D Studio was developed, the 3DS file format, and finally the new MAX file formats came to be that includes the entire scene data as well as each mesh. The unfortunate thing about these formats is that they are way complicated, and can come in either ASCII or binary flavors. So, I'm not going to get into the details of either as I did with PLG or NFF, because we don't have the room right here. What I will do is give a brief description of each and a tiny sample. However, later we will probably end up supporting DXF and/or 3DS.

DXF Files

DXF (Data eXchange Format) files were originally supported by Autodesk's AutoCAD program. There is both a binary and ASCII version of the format, but both share the same name. The format isn't really that bad, but it was never really designed for high-end 3D modeling of solid, lit, textured geometry like we are interested in; this is because the original designers were more interested in modeling 3D mechanical and architectural objects, not aesthetically pleasing imagery. Hence, there is 3D support, but the support isn't complete. Moreover, there's a lot of overhead in DXF files for all the entities it supports, and the sheer amount of technical details that have to be part of a technical drawing that have nothing to do with the object itself. Therefore, we usually have to parse past all the garbage to get to our models. After we get to our 3D model data, reading the file is rather easy.

Format of a DXF File A standard ASCII DXF file has the following main sections:

- `HEADER` section—This contains general information about the model.
- `TABLES` section—This contains definitions of named items.
- `BLOCKS` section—This section contains *block definition* entities describing the entities comprising each block in the drawing.
- `ENTITIES` section—This contains the actual entities that describe the model itself, along with any block references.
- `END OF FILE` marker—Every DXF file ends with the string `EOF`.

And of course, there is going to be all kinds of data in each one of these possible sections. This data is indicated and grouped into classes with what are called group codes. These group codes help you write parsers, because if you just look at the group code, you can quickly determine the context of the data. Table 6.3 lists the basic group codes.

Playrix Ex. 1023, page 619

09 0672318350 CH06 05•02•2003 2:40 PM Page 595

TABLE 6.3 DXF File Group Code Classes

Group Code Range	Meaning
0–9	String value
10–59, 210–239	Floating point value
60–79	Integer value
999	A comment

Whenever there is anything in a DXF file, first there will be a group code to class the data. Thus, for every single number or string in a DXF file, first you will see a group code—if this seems like a lot of overhead and waste, it is—but it makes parsing easier. Going further with group codes, they are subclassed in each range to further detail the data, but keep in mind that 99% of the group codes are irrelevant to us. Table 6.4 lists the group codes in greater detail.

TABLE 6.4 DXF File Group Codes in Detail

Group Code	Meaning
0	Identifies the start of a file, entity or table
1	The primary text value for an entity
2	A name of some kind
3–4	Other textual or information
6	Line type
7	Line style
8	Layer name
9	Variable name identifier, used in the HEADER section
10	Primary x coordinate
11–18	Additional x coordinates
20	Primary y coordinate
21–28	Additional y coordinates
30	Primary z coordinate
31–37	Additional z coordinates
38	Elevation
39	Thickness
40–48	General floating point values
49	Repeat value
50–58	Angles
62	Color number
66	"Entities follow" flag
70–78	Integer values
210, 220, 230	X, Y, Z of an extrusion follow
999	Comments

Playrix Ex. 1023, page 620

09 0672318350 CH06 05•02•2003 2:40 PM Page 596

TIP

I am only going this far with DXF files because when I first learned them about 10 years ago, all the numbers confused me, so I don't want you to have that same feeling. Alas, you at least have an idea of what all the numbers represent now, and why they are all over the place. There seems to more overhead that 3D data!

The group code values always come first, then the "thing." For example, at the start of every DXF file, you will find a string value that indicates the beginning of a file. Referring to Table 6.3 and Table 6.4, string values are `0–10`, and the start of file is `0`, so we would see something like this at the beginning of every DXF file:

```
0
SECTION
```

Suppose you want to add a comment—you would use group code `999` and then put the comment like this:

```
999
This is a comment, stupid huh!
```

Now let's take a look at a generic, empty DXF file. Note that I have added comments using the C++ `//`, but they are for illustrative purposes only, and do not exist in the DXF file format:

```
0 // Begin HEADER section group code
SECTION

2 // group code
HEADER
 // Header variable items go here

0 // group code
ENDSEC  // End HEADER section

0       // Begin TABLES section group code
SECTION

2       // TABLES group code
TABLES

0       // TABLE group code
TABLE
```

Playrix Ex. 1023, page 621

09 0672318350 CH06 05•02•2003 2:40 PM Page 597


```
2       // view port ENTITY group code
VPORT
70

// viewport table maximum item count)
// viewport table items go here>>>>

0      // end the TABLE group code
ENDTAB

0      // TABLE group code
TABLE

2     // group code allows any of these
LTYPE, LAYER, STYLE, VIEW, UCS, or DWGMGR
70
// Table maximum item count
// Table items go here

0      // end TABLE group code
ENDTAB

0
ENDSEC  // End TABLES section

0    // Begin BLOCKS section
SECTION

2    // BLOCKS group code
BLOCKS

// Block definition entities go here

0
ENDSEC //  End BLOCKS section

0      // Begin ENTITIES section
SECTION

2      // and here's where the ENTITIES start
ENTITIES
// Drawing entities go here
```


Playrix Ex. 1023, page 622

09 0672318350 CH06 05•02•2003 2:40 PM Page 598


```
0     // end SECTION group code
ENDSEC // End ENTITIES section

0
EOF // End of file
```

As you can see, there's a lot of stuff in an empty file, but luckily the only thing we are interested in is the `ENTITIES` section, because this is where the 3D data will be. There are many entities as shown in Table 6.5.

TABLE 6.5 Useful DXF File Entities

`ENTITY` **Tag**	**Description**
`LINE`	A line
`POINT`	A point
`CIRCLE`	A circle
`ARC`	An arc
`TRACE`	A trace in space
`SOLID`	A solid object
`TEXT`	A text string
`SHAPE`	A shape
`POLYLINE`	A connected collection of lines
`VERTEX`	A single vertex
`3DFACE`	A 3D face or polygon

We are only interested in the `3DFACE` entity, which is what all the 3D geometry exported using DXF files is stored in. Note however that there's no lighting, texturing, or camera information at all. DXF is very simple mesh data, and that's it. Anyway, the `3DFACE` entity looks like this for a four-sided polygon (I'll use `//` for comments again, but remember that they are illegal; `999` is the real comment group code):

```
0 // this is the start of something,
  // so 0 for the group code
3DFACE
10 // x component of vertex 0 of face
x0 // and here's the value

20 // y component of vertex 0 of face
y0 // and here's the value

30 // z component of vertex 0 of face
z0 // and here's the value
```

Playrix Ex. 1023, page 623

09 0672318350 CH06 05•02•2003 2:40 PM Page 599


```
// now for the next vertex we increment the
// XYZ group codes by 1

11 // x component of vertex 1 of face
x1 // and here's the value

21 // y component of vertex 1 of face
y1 // and here's the value

31 // z component of vertex 1 of face
z1 // and here's the value

// and for the next vertex we again increment
// the XYZ group codes by 1

12 // x component of vertex 2 of face
x2 // and here's the value

22 // y component of vertex 2 of face
y2 // and here's the value

33 // z component of vertex 2 of face
z2 // and here's the value

// and for the next vertex we again increment
// the XYZ group codes by 1

13 // x component of vertex 3 of face
x3 // and here's the value

23 // y component of vertex 3 of face
y3 // and here's the value

33 // z component of vertex 3 of face
z3 // and here's the value
```


NOTE

If you wanted to define a triangle, the last vertex `v3` would have the same values as `v2` (`v0`, `v1`, `v2`, `v2`).

And here is our cube in DXF format (I am using columns and a really small font so you can see the majority of the file at once, because DXF files are written out line by line):

Playrix Ex. 1023, page 624

09 0672318350 CH06 05•02•2003 2:40 PM Page 600


```
0
SECTION
2
HEADER
0
ENDSEC
0
SECTION
2
TABLES
0
TABLE
2
LAYER
70
153
0
LAYER
2
Cube
70
0
62
15
6
CONTINUOUS
0
ENDTAB
0
ENDSEC
0
SECTION
2
ENTITIES
0
3DFACE
8
CUBE
10
1.000000
20
-1.000000
30
```

```
0
SECTION
2
HEADER
0
ENDSEC
0
SECTION
2
TABLES
0
TABLE
2
LAYER
70
153
0
LAYER
2
Cube
70
0
62
15
6
CONTINUOUS
0
ENDTAB
0
ENDSEC
0
SECTION
2
ENTITIES
0
3DFACE
8
CUBE
10
1.000000
20
-1.000000
30
```

```
0
3DFACE
8
CUBE
10
-1.000000
20
1.000000
30
0.000000
11
1.000000
21
-1.000000
31
0.000000
12
1.000000
22
-1.000000
32
0.000000
13
1.000000
23
1.000000
33
0.000000
62
0
0
3DFACE
8
CUBE
10
1.000000
20
1.000000
30
0.000000
11
1.000000
21
```

```
0
3DFACE
8
CUBE
10
1.000000
20
1.000000
30
2.000000
11
-1.000000
21
-1.000000
31
2.000000
12
-1.000000
22
-1.000000
32
2.000000
13
-1.000000
23
1.000000
33
2.000000
62
0
0
3DFACE
8
CUBE
10
-1.000000
20
1.000000
30
0.000000
11
1.000000
21
```

Playrix Ex. 1023, page 625

09 0672318350 CH06 05•02•2003 2:40 PM Page 601


```
0.000000
11
-1.000000
21
-1.000000
31
0.000000
12
-1.000000
22
-1.000000
32
2.000000
13
1.000000
23
-1.000000
33
2.000000
62
0
0.000000
11
-1.000000
21
-1.000000
31
0.000000
12
-1.000000
22
-1.000000
32
2.000000
13
1.000000
23
-1.000000
33
2.000000
62
0
-1.000000
31
0.000000
12
1.000000
22
-1.000000
32
2.000000
13
1.000000
23
1.000000
33
2.000000
62
0
1.000000
31
0.000000
12
1.000000
22
1.000000
32
2.000000
13
-1.000000
23
1.000000
33
2.000000
62
0
```


NOTE

Notice the `8` followed by `CUBE` before each `3DFACE` is defined. This is totally acceptable and the reason why there are group codes—you can just disregard it if you like. However, in this case the `8` is a *layer name* group code, and the `CUBE` string is the name of the layer. A layer in this case is the name of the 3D object I exported, which was called `CUBE`.

As you can see, parsing this file isn't as hard as you think. You simply find the `ENTITIES` section and then start reading each `3DFACE`. The only bummer is that DXF files do not use vertex lists—they use independent vertices for each face.

3DS/ASCII Files

3D Studio Max supports a number of file formats, but its most native formats are 3DS and ASCII. They are actually the same format, but 3DS is binary, and ASCII is human-readable. The formats are extremely robust and you can do just about anything with them, so I'm not going to try to explain them. Rather, I think that if you just look at a file dump of our little 2×2×2 cube located at (0,0,0), you will be able to see how the format works:

Playrix Ex. 1023, page 626

09 0672318350 CH06 05•02•2003 2:40 PM Page 602


```
Ambient light color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Cube"
Tri-mesh, Vertices: 8     Faces: 12
Vertex list:
Vertex 0:  X:-1.000000     Y:-1.000000     Z:-1.000000
Vertex 1:  X:-1.000000     Y:-1.000000     Z:1.000000
Vertex 2:  X:1.000000     Y:-1.000000     Z:-1.000000
Vertex 3:  X:1.000000     Y:-1.000000     Z:1.000000
Vertex 4:  X:-1.000000     Y:1.000000     Z:-1.000000
Vertex 5:  X:1.000000     Y:1.000000     Z:-1.000000
Vertex 6:  X:1.000000     Y:1.000000     Z:1.000000
Vertex 7:  X:-1.000000     Y:1.000000     Z:1.000000
Face list:
Face 0:    A:2 B:3 C:1 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 1:    A:2 B:1 C:0 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 2:    A:4 B:5 C:2 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 3:    A:4 B:2 C:0 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 4:    A:6 B:3 C:2 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 5:    A:6 B:2 C:5 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 6:    A:6 B:7 C:1 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 7:    A:6 B:1 C:3 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 8:    A:6 B:5 C:4 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 9:    A:6 B:4 C:7 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
```

Playrix Ex. 1023, page 627

09 0672318350 CH06 05•02•2003 2:40 PM Page 603


```
Smoothing:  1
Face 10:    A:1 B:7 C:4 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
Face 11:    A:1 B:4 C:0 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:  1
```

Caligari COB Files

Over the years, I have really come to be a strong proponent of Caligari trueSpace. It's a very powerful 3D modeler, and I challenge anyone to argue that its interface isn't the simplest to use in the world. Anyway, the COB format is similar to 3DS format, in as much as what's in the file, and there are both binary and ASCII versions of this format, too. Anyway, the format is much too complex to describe in detail, but once again a listing of our venerable cube shows much more than I can describe in a few pages about the format, so take a look at the following:

```
Caligari V00.01ALH
PolH V0.06 Id 16796788 Parent 0 Size 00000981
Name Cube
center 0 0 1
x axis 1 0 0
y axis 0 1 0
z axis 0 0 1
Transform
1 0 0 0
0 1 0 0
0 0 1 1
0 0 0 1
World Vertices 8
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 -1.000000
1.000000 -1.000000 1.000000
-1.000000 1.000000 -1.000000
1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
Texture Vertices 14
0.000000 0.333333
0.000000 0.666667
0.250000 0.333333
0.250000 0.666667
```


Playrix Ex. 1023, page 628

09 0672318350 CH06 05•02•2003 2:40 PM Page 604


```
0.500000 0.000000
0.500000 0.333333
0.500000 0.666667
0.500000 1.000000
0.250000 0.000000
0.250000 1.000000
0.750000 0.333333
0.750000 0.666667
1.000000 0.333333
1.000000 0.666667
Faces 6
Face verts 4 flags 0 mat 0
<2,2> <0,0> <1,1> <3,3>
Face verts 4 flags 0 mat 0
<4,4> <0,8> <2,2> <5,5>
Face verts 4 flags 0 mat 0
<5,5> <2,2> <3,3> <6,6>
Face verts 4 flags 0 mat 0
<6,6> <3,3> <1,9> <7,7>
Face verts 4 flags 0 mat 0
<4,10> <5,5> <6,6> <7,11>
Face verts 4 flags 0 mat 0
<0,12> <4,10> <7,11> <1,13>
DrawFlags 0
Radiosity Quality: 0
Unit V0.01 Id 16796789 Parent 16796788 Size 00000009
Units 2
Mat1 V0.06 Id 16799524 Parent 16796788 Size 00000090
mat# 0
shader: phong  facet: auto32
rgb 1,1,1
alpha 1  ka 0.23  ks 0.81  exp 0.91  ior 1
ShBx V0.00 Id 16799525 Parent 16799524 Size 00000627
Shader class: color
Shader name: "chrome" (chrome)
Number of parameters: 3
base colour: color (255, 255, 255)
vector: vector (0, 1, 0.5)
mix: float 0.5
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Shader class: reflectance
```

Playrix Ex. 1023, page 629

09 0672318350 CH06 05•02•2003 2:40 PM Page 605


```
Shader name: "caligari phong" (caligari phong)
Number of parameters: 8
ambient factor: float 0.23
diffuse factor: float 0.19
specular factor: float 0.81
exponent: float 43.8135
specular colour: color (61, 67, 255)
transmission factor: float 0
mirror factor: float 0.78084
refraction: float 1
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
END  V1.00 Id 0 Parent 0 Size        0
```

As you can see, the beef is up at the top, but basically it's once again a vertex-based format with a nice face list. Also, it's darn near in plain English, so you almost don't even need to read a DOC on the format—you can try mucking with an object a bit and figure out the majority of the format in a couple hours, as far as orientation, lights, texture coordinates, and so forth, go.

Additionally, trueSpace has what is called the .SCN format, or "Scene," which supports entire scenes and worlds. 3D Studio Max supports this too, with the .MAX format. Thus, we can potentially use either modeler (or Maya or Lightwave for that matter) to not only define objects, but also entire worlds with lights, and so on. So as I said before, when you get ready to write a game, see whether you can leverage a 3D modeler along with some custom plug-in code, and use it as your game editor. Most of the time this approach works great, and 3D modeler developers are making it easier each day, with more and more support for game designers and editors with each iteration of their tools.

Icon Cool

3D Studio Max has a scripting language called MaxScript. With MaxScript, you can pretty much access anything you want in a 3D world. This way you can write scripts or macros that add functionality to 3D Studio Max, and you can basically turn it into a game editor!

Microsoft DirectX .X Files

The light dims … Last but not least is Microsoft's DirectX format. I am not an expert on this format, and I haven't used it for much, other than exporting it and letting Direct3D import it for me. That is, I have never written a parser. But I can tell you one thing—this format has everything but the kitchen sink. It's very complex, but also very powerful—definitely not something we are even going to think about writing a parser for. Therefore, we aren't going to touch .X files ourselves. But the 3D acceleration portion of the book (on the CD) loads and uses them at the API level, if you're interested.

Playrix Ex. 1023, page 630

09 0672318350 CH06 05•02•2003 2:40 PM Page 606

3D File Format Summary

As you can see, various 3D file formats each have different pros and cons, but definitely anything that is human-readable (has an ASCII version) is a plus. For more information on 3D file formats, try *3D Graphics File Formats* by Keith Rule (Addison-Wesley), or searching on the Net is always a good thing. For example, here are two nice URLs for a list of file format descriptions:

```
http://www.informatik.uni-frankfurt.de/~amueller/FF02.HTM
```

or

```
http://astronomy.swin.edu.au/~pbourke/3dformats/
```

However, we will write a parser for each format we need for the book.

NOTE

The majority of file formats we've reviewed export the coordinates in a right-handed system. However, depending on the tool, this can be overridden. But I want to emphasize that it's an issue when we finally write parsers, and if for some reason down the road you load an object into your game that supports some specific file format and everything is inside out, you simply need to invert the order of vertices and flip the z-axis in most cases.

Basic Rigid Transformations and Animation

If you're reading this book, you should have a fairly good grasp on 2D animation techniques such as translation, bitmap scaling, and rotation. However, in 3D there's a lot more we can do with 3D objects, because we have a total of six degrees of freedom to move objects around, and we can deform the meshes themselves.

We've already talked a little about *key frame animation* and *motion data-based animation*, and we will actually implement these types of mesh animations later in the book when we talk about character animation. Right now, I want to cover some of the basic animation techniques to simply move your 3D objects around and change their orientation, so that you can at least create games that have rigid objects that move around and do things. So, let's take a brief look at how to approach the basic animation techniques of motion, rotation, and morphing.

3D Translation

Translation in 3D is trivial, and we have already covered both the manual and matrix derivations for it, so I won't belabor the mechanics of it. What I want to talk about is how to translate an entire mesh, or better yet, the best way to translate a mesh in a 3D game.

Because each object has a set of local or model coordinates that are all relative to a local origin of (0,0,0), we don't really want to translate the local coordinates of an object to

Playrix Ex. 1023, page 631

09 0672318350 CH06 05•02•2003 2:40 PM Page 607

translate it. The better approach is to have the position of the object and then use it as the reference point during the local-to-world transformation to translate the object—which we have already done. In fact, our object already has a field to track the position of the object that it represents, called `world_pos`. Thus, we leave the actual translation of every point in the mesh to the local-to-world transform and we only have to deal with translating a single point to move an entire object! Therefore, if you want to move an object by the vector t = <tx, ty, tz>, all you need to do is this:

```
object.world_pos.x+=t.x;
object.world_pos.y+=t.y;
object.world_pos.z+=t.z;
```

Relatively simple. Okay, now that we know we can do everything with a single translation to a single point, the next question is how to move objects in 3D? Well, this is really more of a question that applies to AI. Your AI will generate a proper stream of motion or translation values. But just for fun, let's try to make an object move around in a circle in 3D. All we need to do is compute either absolute position values or relative position values from the last position. To makes things simple, let's use absolute values rather than relative. Let's say that we want to move an object on a elliptical path in a plane that is parallel to the x-z plane. Here's the code that would do it:

```
float x_pos = a*cos(angle);
float y_pos = altitude;
float z_pos = b*sin(angle);
```


You would just plug in values for `a`, `b`, `angle`, and the altitude, and the math would outpost the positions of an elliptical path—too easy, huh? Okay, what about a spiral that spirals up each cycle—that's easy, too:

```
// initial conditions
float altitude = 0;
float rate     = .1;

// this code is iterated each cycle
float x_pos = a*cos(angle);
float y_pos = altitude+=rate;
float z_pos = b*sin(angle);
```

And that's it—the point `(x_pos, y_pos, z_pos)` will spiral up from the ground plane y = 0. As you can see, writing motion code for 3D is just as simple as it is for 2D, and we are even using curves here without any problem at all.

3D Rotation

Rotation is a bit trickier than translation. Math isn't the hard part (although it's much more complex), but the tracking of the orientation is more difficult. For example, when

Playrix Ex. 1023, page 632

09 0672318350 CH06 05•02•2003 2:40 PM Page 608

you move an object to (x,y,z), you know it's at (x,y,z), simple as that. On the other hand, when you rotate an object (ang_x, ang_y, ang_z), the orientation isn't necessarily (ang_x, ang_y, ang_z) relative to the axis—remember all that stuff about gimbal lock? Additionally, when you rotate an object, the order that you apply the rotation matters; that is, XYZ is completely different from YXZ, and so forth. Of course, if you are rotating an asteroid around and you create a rotation rate (rx, ry, rz) with which you rotate the asteroid each frame, who cares? But if you are rotating a tank or ship around, you need to be able to compute the orientation of the ship when you need it. Hold onto that thought for a moment.

To rotate an object in 3D, you always want to rotate the local coordinates of the object. You want to perform the rotation transformation in the local space with (0,0,0) being the object's local origin (and not in world space—otherwise your object will rotate off center). Now it's really up to you if you want to perform the rotation and then store the rotated coordinates back into the local coordinate storage, or if you want to make the current orientation of each object part of the viewing transform. That is, when an object is getting ready to be rendered, some code says to rotate the objects a certain angle and then pass it on in a temporary array. The pro of this technique is that your model never gets mangled, and it can be used as a master copy and other objects can just link to it. The downside to this technique is that for an object that isn't animating or that is fairly stable, it will have to go through a rotation transformation each cycle, even if it's not animating—wasting cycles.

Keeping that in mind, if you want to rotate an object, you would simply transform its local coordinates in place and then store the results back. For example, let's say that we have a tank simulation, and a tank object called `tank1` that we already have defined and loaded in is sitting there oriented straight down the +z axis. As the player turns right or left, we can rotate the model in local coordinates:

```
// initialization, assume tank drives around in
// x-z plane therefore rotation to turn is parallel
// to the y-axis, that is yaw
tank1.dir.x = 0;
tank1.dir.y = 0;
tank1.dir.z = 0;

// game code
if (turn_right)
   {
   for (int index=0; index<tank1.num_vertices; index++)
       {
       // rotate the point, assume the call does it
       Rotate_Point_On_Y(&tank1.vlist_local[index], -5);
```

Playrix Ex. 1023, page 633

09 0672318350 CH06 05•02•2003 2:40 PM Page 609


```
        // update the current orientation, so we know
        // where the tank is pointing
        tank1.dir.y-=5;
        } // end for

    } // end if
else
if (turn_left)
    {
    for (int index=0; index<tank1.num_vertices; index++)
        {
        // rotate the point, assume the call does it
        Rotate_Point_On_Y(&tank1.vlist_local[index], 5);

        // update the current orientation, so we know
        // where the tank is pointing
        tank1.dir.y+=5;
        } // end for

    } // end if
```

Not much is going on in the code; if the player turns left, the entire vertex list is rotated parallel to the y-axis +5 degrees, and if the player turns right, the vertex list is rotated –5 degrees. However, here's the clincher—after you rotate a 3D object, you have no idea what its orientation is by just looking at the mesh.

Therefore, this information is exactly what is tracked in the direction field `object.dir` via the updates to `tank1.dir`. At any time, we can look at `tank1.dir` and know that the tank is pointing at that angle relative to the positive X-axis (which we assume is the 0-degree direction for y-axis rotation).

This method works great, but when you start rotating all three axes at the same time, it kind of falls apart. As I said, the actual orientation of an object might not be the sum of the rotations to the individual axes taken one at a time. However, there's a better way to track this—using an orientation vector. That is, for every object in your 3D world, when you load it and it's oriented toward the +z axis, you initialize a unit 3D vector that is also pointing down the 3D +z. Whenever you transform the 3D object, you transform the unit direction, too. This is shown in Figure 6.52. After 1,000,000 rotations, if you review the rotation vector, it will always reflect the current direction of the object. This is what the `dir` field is also used for. Alas, there's a catch. What if you rotate the object parallel to the z-axis and flip it upside down? The direction vector is still correct, but you have lost information because a vector has no "roll" tracking (but quaternions do!).

Playrix Ex. 1023, page 634

09 0672318350 CH06 05•02•2003 2:40 PM Page 610

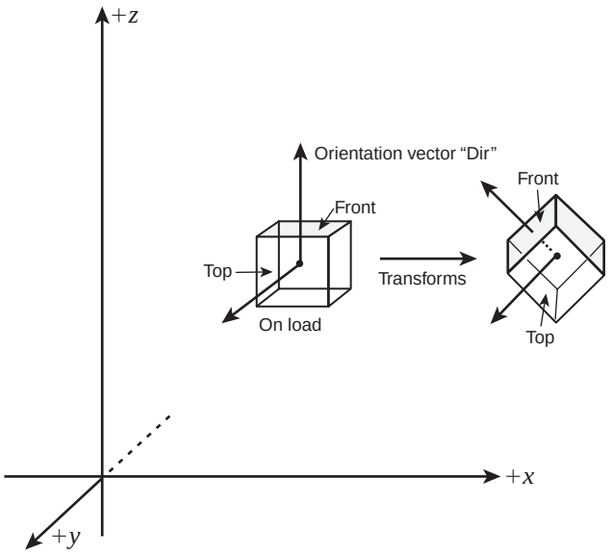


FIGURE 6.52 Tracking the orientation of an object after loading and transformations.

The point is that if you want absolute orientation tracking via a vector, you need three vectors inside your object structure/class, shown in Figure 6.53. You initialize these vectors to be the x-y-z local axis of the object, and then when you rotate the object, you rotate all three of the direction vectors. You will always be able to determine the absolute orientation of your object, because whenever you perform a rotation to the object, you perform the same rotation to the local axis, and thus they always reflect the transformations.

3D Morphing

The final simplistic type of animation that you can perform to a 3D mesh is morphing. I don't mean morphing from one mesh to another, but more scaling and shearing the mesh data in real time to make something look alive. For example, you could write the following code to make something look like it's breathing:

```
int up_down = 1; // -1 means get smaller
                 // 1 means get bigger

float scale = 1; // current scale

while(1)
{
// scale up or down?
if (up_down==1)
```

Playrix Ex. 1023, page 635

09 0672318350 CH06 05•02•2003 2:40 PM Page 611


```
   {
   Scale_Object(&object, 1.05);
   scale*=1.05;
} // end scale
else
   {
   Scale_Object(&object, .95);
   scale*=.95;
   } // end else

// test if we are finishing cycle and changing
// directions
if (scale < .75 ¦¦ scale > 1.25)
   up_down=-up_down;


} // end while
```

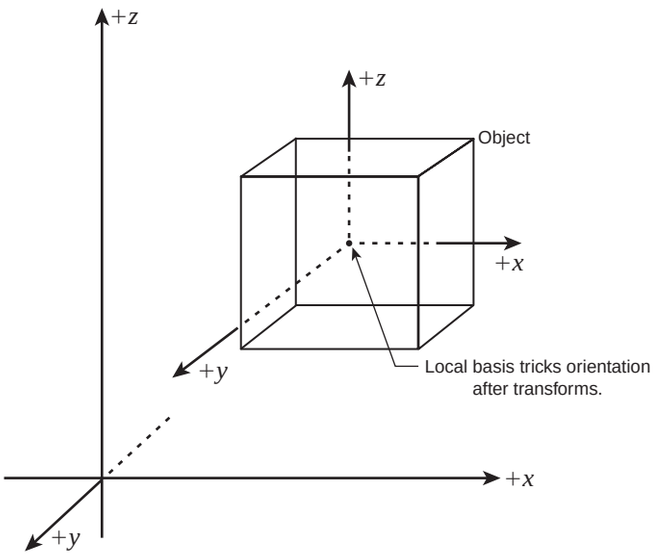


FIGURE 6.53 For total control of object orientation, an entire local axes basis must be stored.

The preceding code will scale an object by 5% up or down each cycle, making it look like it's breathing, or whatever. Of course, we have to write the function `Scale_Object()`, but that's nothing more than iterating through the sent objects vertex list and multiplying by the scale factor—easy. The point that I want to make is that we are assuming that we are

Playrix Ex. 1023, page 636

09 0672318350 CH06 05•02•2003 2:40 PM Page 612

modifying the local coordinates each cycle. This is fine, but slowly numerical errors will creep into the model, and it will deform hundreds, thousands, or tens of thousands of cycles, depending on the data. This is also true in the rotation animation.

CAUTION

Anytime you modify the local coordinates, there exists the possibility that after many modifications the original mesh data will be lost forever because of numeric errors, or *drift*.

The only real solution to this problem is to either keep a master copy of each mesh and refresh all objects every so often, or to make the rotation and morphing operations on the fly during the pipeline and apply them to the local coordinates of the object in question—but the results are stored in a temporary area, rather than overwriting the original mesh data.

Review of the Viewing Pipeline

We have covered so much ground once again. I'm cringing at this chapter's page count on MS Word, but hey—we aren't making grilled cheese sandwiches! Anyway, now that you know a little about all the parts of the 3D pipeline, let's recap the pipeline and try to fill in as much as possible, annotating the pipeline in places where we know we have lots of work to do.

In Figure 6.54, we see a fairly detailed view of the 3D pipeline, with all kinds of stuff hanging from it. The major parts are the following:

- Local transformations and animation—This stage is where the local coordinates of each object are transformed. There might be rotation, scaling, or other vertex operations that are performed in place.
- Local-to-world transformation—This stage is where the local coordinates are translated to world coordinates one at a time based on the position of the object, which is tracked relative to its local center at `world_pos` (in our object definition).
- Hidden surface removal and lighting—This stage is where we remove back-faces from the 3D models, as well as remove entire objects from the 3D pipeline. Additionally, lighting calculations are usually performed in world coordinates.
- Object removal—This stage is performed separately sometimes, and can be unlinked to the lighting phase. The only reason that lighting and hidden surface removal are performed at the same time is that surface normal calculations are made for the hidden surface removal that can be reused by the lighting models.
- World-to-camera transformation—This stage is where the final geometry that is potentially visible is transformed relative to the camera position and view angles.
- 3D clipping—This stage is where polygons are clipped to the 3D viewing volume.

Playrix Ex. 1023, page 637

09 0672318350 CH06 05•02•2003 2:40 PM Page 613

- Perspective projection and clipping—This stage transforms the camera coordinates so that the objects have perspective. Additionally, the perspective transformation can normalize the viewing volume into a unit cube or cuboid shape, which makes 3D clipping easier to perform. Hence, sometimes 3D clipping is performed after the perspective transform.
- Screen projection—This is nearly the last stop in the pipeline. Here the view plane coordinates are mapped to the screen and passed to the rasterizer for final rendering.
- Rasterization and clipping—This is the last stage in the 3D pipeline. Here a list of polygons, vertex colors, texture maps, and so forth are passed to a rasterizer that just blasts the polygons down on the screen. Additionally, some polygons might extend past the edges of the screen (if they weren't clipped in a previous stage), and will be clipped at the pixel or scan line level.

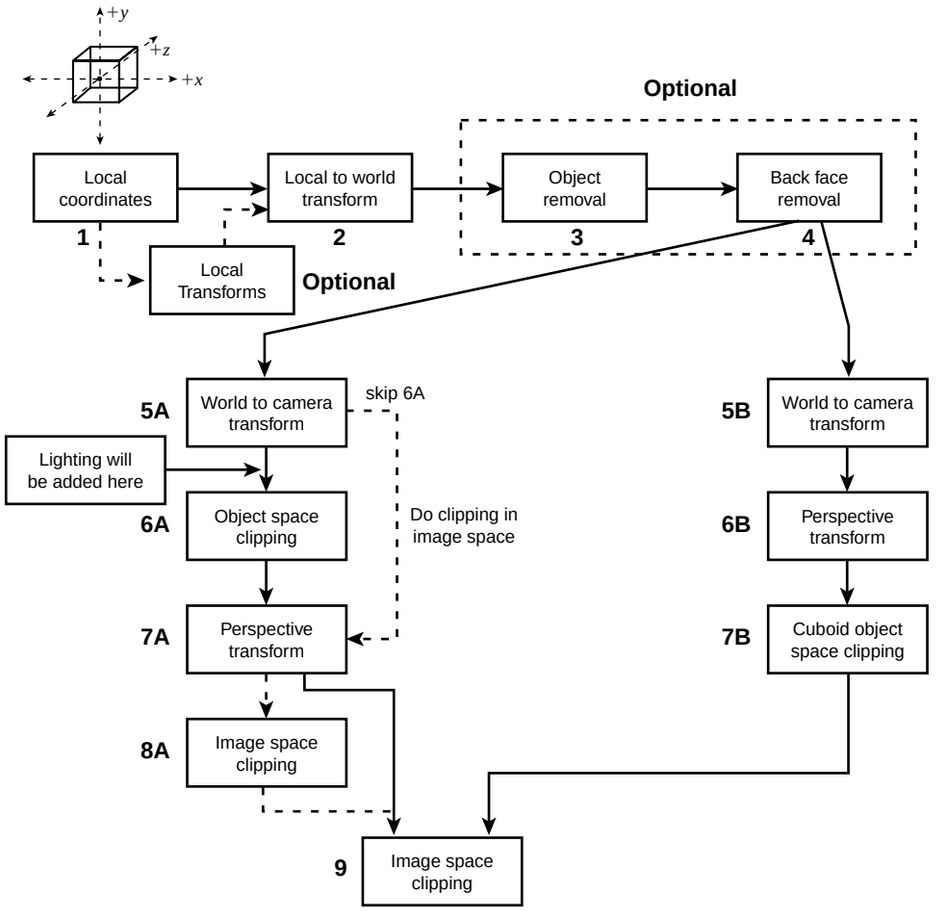


FIGURE 6.54 The final 3D pipeline review.

Playrix Ex. 1023, page 638

09 0672318350 CH06 05•02•2003 2:40 PM Page 614

That's not a bad first shot at the 3D pipeline. Granted, there are a lot of details involved in making it work, but for the most part, that's it. Although we haven't talked much about lighting, you can imagine what's involved: lots of math, light sources, and the orientation of the polygons—but we will get to that in the upcoming chapters.

Types of 3D Engines

Now I want to take a moment to talk about the types of 3D engines out there. I don't want to get too much into the technology, but I just want you to get a feel for them—why they exist, and what they are each good at doing.

Space Engines

Figure 6.55 depicts a view of a good old space game: *Star Wars: X-Wing*. As you can see, the majority of the game is made out of objects floating in space. Games of this type are typically much simpler to write than first-person shooters for a number of reasons, including animation, collision detection, and database issues. However, in the most basic terms, space games are easier to write because the 3D engines aren't as complex. In most cases, 3D space games are object-based, meaning that all entities are objects, and during rendering, many of them are discarded from the pipeline. Then during the rendering phase, a simple painter's algorithm or Z-buffer is used to sort the polygons that make up each object. However, each object can be easily sorted against the other objects, because they represent ships in space.

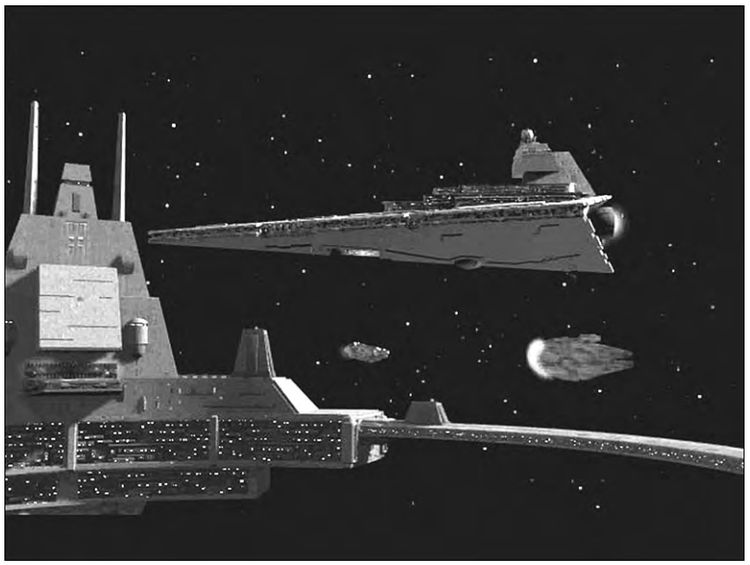


FIGURE 6.55 A screenshot of *Star Wars: X-Wing*.

Playrix Ex. 1023, page 639

09 0672318350 CH06 05•02•2003 2:40 PM Page 615

Additionally, most space games these days are written with polygon engines. However, some of the first 3D space games like *Wing Commander* were written using sprite engines, with lots of prerendered views, along with fast bitmap reflection and scaling algorithms. The positions of the ships and so forth were all represented by single points in 3D—so the engines were 3D in a sense, but the models were not. Anyway, space games these days are written with polygon engines for the most part, and lighting, animation, and so on are all handled in the normal manner.

The hard thing about space games is the motion. In a first-person shooter, it's very hard to animate bipeds walking around, but many times you have prerecorded animation files and you are usually walking on a 2D surface, like a floor. The problem with 3D space games is not so much the rendering, but the motion and AI for space ships. This is a bit more challenging, because the ships have to be able to make smooth turns, rolls, and operate in 3D. This means the use of 3D curves might have to be employed, along with other techniques. But as a rule of thumb, your first 3D games should be ground-based (in a desert, or in space) to simplify the rendering algorithms and database work.

Terrain Engines

Terrain engines are the next step up from space engines. A terrain engine can of course have more than terrain, but the terrain is a major component of the world, and a proper method of determining visible polygons is a must. Figure 6.56 is a screenshot from the game *Tread Marks* by Longbow Digital Arts, which is a 3D game with a lot of terrain. Aside from the terrain, the objects in a terrain (or any terrestrial) game are fairly easy to deal with. They can move around over the terrain (maybe some fly over), and in most cases are tanks or vehicles, so their animations aren't that bad. Of course, if you are making a mech warrior game, you have to deal with complex animation and biped terrain tracking. But the major problem with terrain engines is how to represent the world database, which can be quite large.

For example, suppose that you want to create a world that is a mesh of polygons 100,000×100,000 units in size, with each polygon no more than 200×200 units. That means you have a polygon mesh that is of the order

```
(100000/200)*(100000/200) = 250,000 polygons!
```

Playrix Ex. 1023, page 640

09 0672318350 CH06 05•02•2003 2:40 PM Page 616



FIGURE 6.56 A screenshot of the most excellent game *Tread Marks*.

See what I mean?—that's a lot of polygons to track. Therefore, you have to use some kind of partitioning scheme to keep the number of active polygons that could potentially be visible at any time down to a reasonable minimum. One such technique is a *sectorizing* of the world. For example, take a look at Figure 6.57. Here I have sectorized our 100,000×100,000 game grid into an 8×8 matrix of sectors, each consisting of roughly 1,000 polygons per sector (250,000/256), which is manageable. And because I can only see maybe 1,000 units in front of me, I am guaranteed to only have to look in four sectors at once (worst case at a corner point), which means that at most I will have four sectors, or 4,000 polygons, to deal with at any one time. Not bad. Of course, that's minus objects.

Again, terrain or land-based games are a little more complex than space games because of the terrain and large database, but other than that, the rendering can be done using a modified painter's algorithm or binary space partition (if you want to get really big worlds), or a plain old Z-buffer. But you can get away with very little. Of course, another hard part about terrain engines is the motion of the objects and their tracking of the terrain contour. This is *terrain following*, nothing more than keeping your game objects oriented so their legs, feet, wheels, or whatever stay above ground, and that their center of masses up vectors are parallel to the terrain they are moving on.

Playrix Ex. 1023, page 641

09 0672318350 CH06 05•02•2003 2:40 PM Page 617

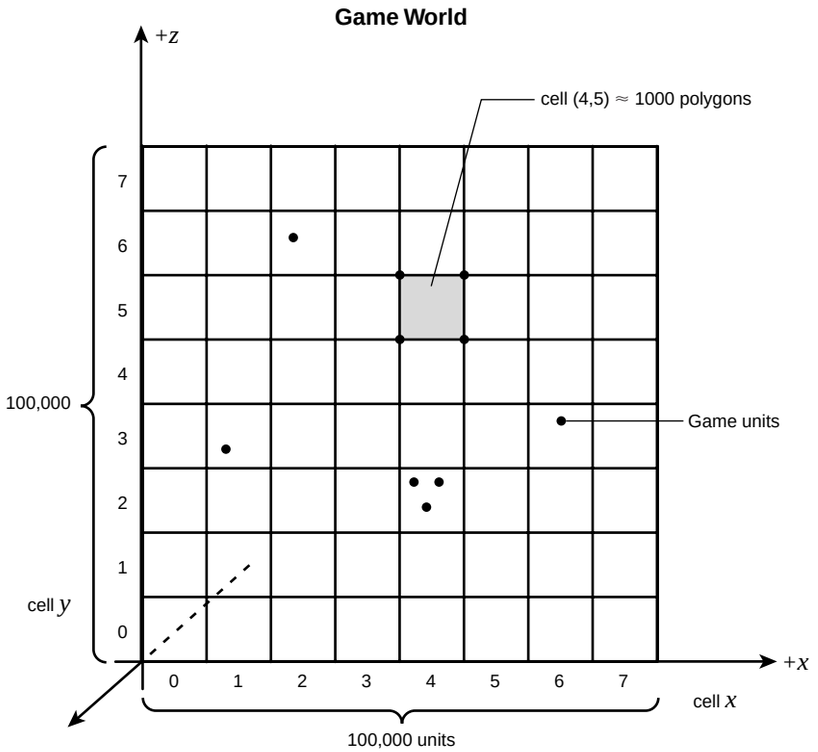


FIGURE 6.57 Sectorizing a large world into smaller blocks.

FPS Indoor Engines

These are the bad boys. Figure 6.58 is a screenshot of one of the quintessential 3D first-person shooter (FPS) games available: *Unreal Tournament*. First-person shooters are really the hardest games to write, for a number of reasons. First, you are indoors many times, and that means high detail and close quarters. Secondly, the worlds for FPS games are getting huge, and hence so are the polygon databases. This means that there is simply no way to just throw the whole world at the 3D pipeline—you have to sectorize your world using visibility schemes to minimize the number of polygons taken into consideration.

This means you'll need octrees, binary space partitions, portal engines, or whatever. In a space or terrain outdoor game, you can get by with the 3D models not looking right, polygons sorted incorrectly or missing, or whatever. But in a FPS game, you simply can't do that. Referring again to Figure 6.58, these games are solid as a rock, and really immerse the player in the worlds. Players simply won't accept an unstable, shaky, or ugly world. These things have to look real!

Playrix Ex. 1023, page 642

09 0672318350 CH06 05•02•2003 2:40 PM Page 618



FIGURE 6.58 3D first-person shooter *Unreal Tournament*.

Finally, the physics modeling and AI for a FPS are very complex, again because of the intimate nature and close proximity of the games. The players "see" everything very well, and know what to expect. When a space ship blows up in a space shooter, that's not too hard to simulate, but when you fire a pulse at a player and he falls in the water, players want to see the water ripple and the box floating in it to bob up and down!

Ray Casters and Voxel Engines

So far we have talked about polygon-based engines, but there are other types of 3D engines based on ray casting and voxels. *Ray casting* is the technique used in many of the first 3D FPS games, like *Wolfenstein 3D*. These games are based on *forward ray tracing*, which is tracing a ray from the viewpoint of the player through the view plane until it hits a single object, as shown in Figure 6.59. Based on the geometry of the world and hints about the object, a 3D scene can be generated very quickly using this technique.

NOTE

Actually, a programmer named Paul Edelstein created the first 3D ray caster I ever saw on an Atari 800 game in the late 80s, but it was too ahead of its time for anyone to realize the power of the technique.

Playrix Ex. 1023, page 643

09 0672318350 CH06 05•02•2003 2:40 PM Page 619

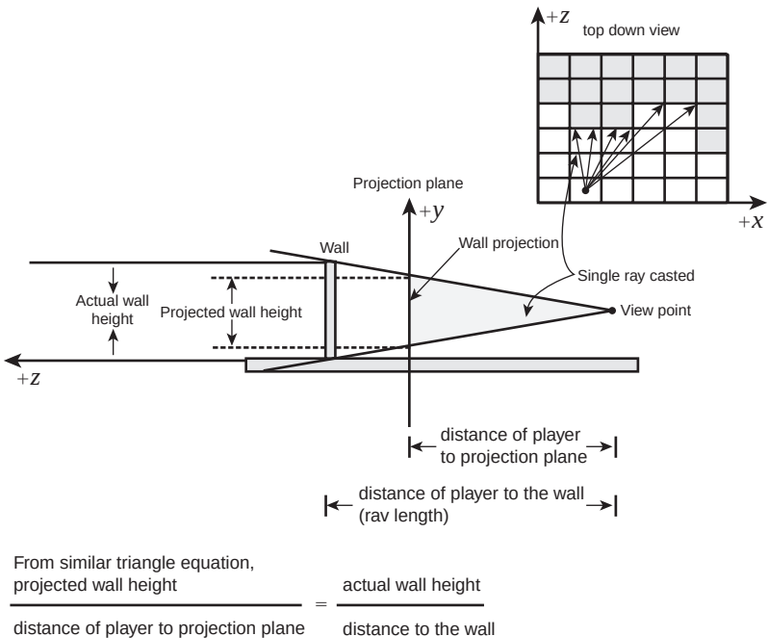


FIGURE 6.59 Ray casting mechanics.

The problem with ray casting is its limited capabilities. However, I believe that we will once again revisit advanced ray casters that perform a kind of ray casting/ray tracing hybrid in the next five years that will render in a more realistic way than with polygons—my two cents. Anyway, the other ray casting type technique is called *voxel graphics*, or *volumetric rendering*. One of the most notable games to use voxel graphics is Novalogic's *Comanche*. Technically, real voxel graphics are really not based on ray casting, but a technique called the *marching cubes algorithm*, which is used to render volumetric data. A 3D data set is used to render a 3D image—hence, the data is one-to-one with the image we desire to render. Medical imaging uses voxel systems. However, we have found that we can take voxel data and use other tricks based on ray casting to render the images very quickly. If you play with my voxel demo on the CD, you will see how fast it is.

The problem with ray casters and voxel engines is that you need a lot of data, and I mean a lot, to represent anything with detail. Alas, they aren't used that much in games these days, because polygons with texture maps are more efficient and look better. But as I said, I think we are going to see a comeback because ultimately all rendering will be based on light and ray tracing, and ray casting and voxel engines work well in these areas.

Playrix Ex. 1023, page 644

09 0672318350 CH06 05•02•2003 2:40 PM Page 620

Hybrid Engines

The last type of engine I want to talk about is the hybrid. All this means is an engine that is designed to simulate space, land, and FPS all at the same time. This can also mean a single engine that uses different techniques in the same environment. For example, *Wolfenstein* uses ray casting for the world and sprites for the objects. *Doom* uses polygons for the world and sprites for the objects. *Quake* uses polygons for the world and some polygon models for the objects, and some sprites for others, and everything in *Quake III* is based on polygons.

The point is, many times you might want to create a game that has multiple environments, and it might behoove you to use a number of different engine types, so that each is better suited for the task, rather than using one and trying to make it work for everything. As another example, many 3D games these days have particle systems. Many times the particle systems are based on polygons that are *billboards*; that is, polygons that always face the viewer—hence, they are really like 3D sprites.

In conclusion, we are going to stick to full polygon 3D engines. If you're interested in some of the other techniques, maybe to make specific objects in your game easier to implement or whatever, there are a lot of other books on the subjects of sprite engines, ray casting, and other techniques.

Integrating Everything into a Final Engine

We have covered a lot of topics in this chapter, written a lot of pseudo-code, and reviewed lots of examples. At this point, we are ready to actually start writing working code that will enable us to create a crude 3D world, place objects in it, move the objects around, and view the world from a camera. We will do all this in the next chapter, and then each chapter thereafter will focus on specific topics (some that we have already discussed) and add lots of detail to each. However, there is a teenie-weenie bit of code we wrote in this chapter having to do with objects and data structures, and you can find it, along with all the stuff from the next chapter, in `T3DLIB5.CPP¦H`.

So that's really it. No demos at all in this chapter, just a lot of high-level abstract ideas with some examples. Now I'm going to assume you know this stuff and pick up the pace a bit, so if you aren't completely comfortable with everything, please review this chapter again.

Summary

We have without a doubt covered a lot of ground in this chapter, and you probably have more questions than when you started. As we continue on, all will hopefully become clear. Our goal in this chapter was to learn coordinate systems in detail, because we are going to build on it directly in the next chapter, and additionally to get a general overview of everything 3D, just to give you something to think about as you learn each topic in detail. Like I said, 3D is so complex that to learn it, you need to start at the beginning, middle, and end all at the same time!

Playrix Ex. 1023, page 645

CHAPTER **7**

Rendering Wireframe Worlds in 3D

"Would you say I have a plethora of piñatas?"

—El Guapo, *The Three Amigos*

Finally, we're ready to start writing 3D applications! You now have all the fundamental mathematical knowledge and at least a very general idea of all the parts of the 3D engine. In this chapter, we're going to write a simple wireframe 3D engine. As the book progresses, we'll upgrade to a flat-shaded engine, add lighting, texture mapping, and finally end up with something like a *Quake* software engine. Therefore, we're going to move in nice clean steps. Unlike the last chapter, this one is going to be code heavy, and we're going to explore a couple different ways to do everything. Moreover, we're going to be doing some more camera work in this chapter and discuss another camera model, called the UVN system, in addition to the Eulerian model. Here's the hit list:

- General wireframe engine architecture
- Writing a simple 3D file loader
- Building the 3D pipeline
- The local-to-world transform
- Camera models
- The world-to-camera transform
- The camera-to-perspective transform

IN THIS CHAPTER

- General Wireframe Engine Architecture
- Writing a 3D File Loader
- Building a 3D Pipeline
- Rendering a 3D World
- 3D Demos

10 0672318350 CH07 05•02•2003 2:42 PM Page 622

- The perspective-to-screen view port transform
- Object removal
- Back-face removal
- Simple 2D/3D clipping
- Rendering the 3D imagery

General Wireframe Engine Architecture

Instead of just jumping into writing our first 3D engine, let's first lay the foundation of what we want to end up with. Remember, it's pointless for me to just give you a *Quake* engine and say, "here." The point of this book is to iterate version after version and for you to understand every single detail of building a 3D engine from the ground up. This also means making mistakes or simplifications for the sake of doing it.

TIP

In reference to the preceding statements about making mistakes or oversimplifications to make something understandable: One of the greatest simplifications in science is the model of an atom. During high school and even lower-level courses in college, the model of an atom is a nucleus composed of neutrons and protons along with a number of electrons orbiting in nice circular or elliptical orbits. Of course, this isn't true at all. Electrons hardly orbit; in fact, they're nothing more than a probability cloud and by their very nature, we can't locate them with certainty in both their momentum and position. But, for the purposes of many simple experiments, the orbital model works. In that same vein, many of the 3D techniques we'll learn can be improved on, but if you just see the end result, you're doomed to repeat history and make the same kind of mistakes if you don't learn why or when something is better than something else.

In the previous chapter, we covered a great deal of mathematical theory about 3D graphics. The fact is that whether you realize it or not, you know everything you need to about 3D graphics to make a fairly good engine (except maybe some rasterization techniques). So, what we're going to do is take all the mathematical and algorithmic concepts that we covered in relation to coordinate systems, projections, and data structures to build a simple wireframe 3D engine. This engine will allow polygons to be defined either independently or via higher-level objects loaded off disk and then rendered. Additionally, I've written all kinds of transformation functions (a plethora in fact), so we can create a number of different 3D pipelines to experiment with. For example, we might load an object and use hard-coded transforms all the way down the pipeline and render the object as a set of polygons. On the other hand, we might uses matrices to perform the various pipeline transformations and "rip" each object into its constituent polygons, and then render the polygons without caring about the original object(s) that they represented.

Playrix Ex. 1023, page 647

10 0672318350 CH07 05•02•2003 2:42 PM Page 623

The point is that we're going to see more than one way to accomplish the same goal (and if you're an X Window System programmer, you'll feel right at home!). Very quickly, you'll learn that some basic 3D axioms, such as matrices, are beautiful but completely ridiculous when used to perform the local-to-world transform and so forth (with software engines, at least). Finally, we're going to explore two different camera models. The first is the simple Euler (or Eulerian) camera, which is nothing more than a position coupled with a set of rotation angles. The second camera (which we've covered descriptively, but didn't implement) is the UVN system, which is more like what a conventional movie director thinks in terms of; that is, a direction and focal point, along with an orientation relative to that direction.

By the way, if you don't own an advanced TI (or Casio) graphing calculator, buy one. I can't tell you how much it's been helping with the mathematics of the book and testing. Moreover, you can write games on them because many of them have 68000 32-bit processors in them!

The Data Structures and 3D Pipeline

There's not much to our first 3D engine. We're going to build it up as the chapter progresses, but right now, I want to at least lay out a rough diagram of the data sources and 3D pipeline. Figure 7.1 shows that the engine is based on three unique data structures:

- A structure for a general polygon based on a vertex list
- A structure for a general self-contained polygon
- A structure for objects that are composed of a number of polygons

These are implemented with the same data structures from the previous chapter. I've copied them here so you can refresh your memory (and mine—I'm going nuts naming things clearly):

```
// a polygon based on a vertex list
typedef struct POLY4DV1_TYP
{
int state;  // state information
int attr;   // physical attributes of polygon
int color;  // color of polygon

POINT4D_PTR vlist; // the vertex list itself
int vert[3];    // the indices into the vertex list

} POLY4DV1, *POLY4DV1_PTR;
```

Playrix Ex. 1023, page 648

10 0672318350 CH07 05•02•2003 2:42 PM Page 624


```
// a self contained polygon used for the render list
typedef struct POLYF4DV1_TYP
{
int state;  // state information
int attr;   // physical attributes of polygon
int color;  // color of polygon

POINT4D vlist[3]; // the vertices of this triangle
POINT4D tvlist[3]; // the vertices after transformation if needed

POLYF4DV1_TYP *next; // pointer to next polygon in list??
POLYF4DV1_TYP *prev; // pointer to previous polygon in list??

} POLYF4DV1, *POLYF4DV1_PTR;
```

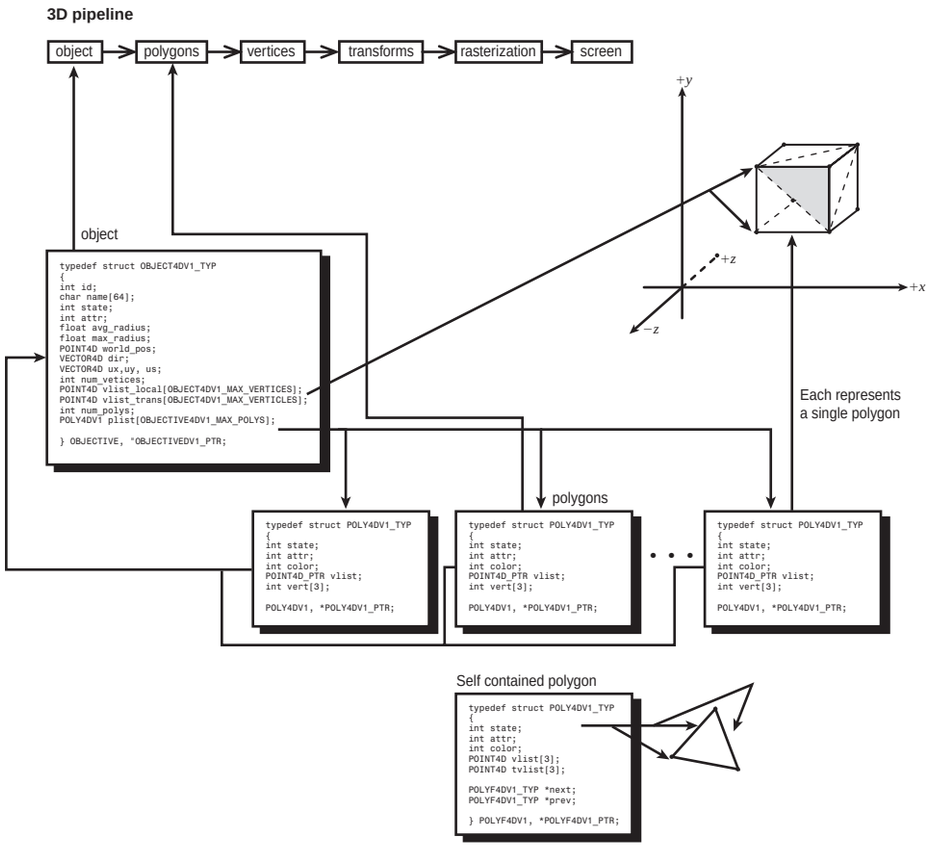


FIGURE 7.1 Data structures of our 3D engine.

Playrix Ex. 1023, page 649

10 0672318350 CH07 05•02•2003 2:42 PM Page 625

Here are some of the #defines for the polygon structures to make life easier on us:

```
// defines for polygons and faces version 1

// attributes of polygons and polygon faces
#define POLY4DV1_ATTR_2SIDED       0x0001
#define POLY4DV1_ATTR_TRANSPARENT     0x0002
#define POLY4DV1_ATTR_8BITCOLOR      0x0004
#define POLY4DV1_ATTR_RGB16        0x0008
#define POLY4DV1_ATTR_RGB24        0x0010

#define POLY4DV1_ATTR_SHADE_MODE_PURE   0x0020
#define POLY4DV1_ATTR_SHADE_MODE_FLAT   0x0040
#define POLY4DV1_ATTR_SHADE_MODE_GOURAUD 0x0080
#define POLY4DV1_ATTR_SHADE_MODE_PHONG  0x0100

// states of polygons and faces
#define POLY4DV1_STATE_ACTIVE       0x0001
#define POLY4DV1_STATE_CLIPPED      0x0002
#define POLY4DV1_STATE_BACKFACE      0x0004
```

Some of the symbols may seem a bit alien, but they'll make sense as we build up the engine. Moving on, here's the object data structure:

```
// an object based on a vertex list and list of polygons
typedef struct OBJECT4DV1_TYP
{
int id;      // numeric id of this object
char name[64];   // ASCII name of object just for kicks
int state;    // state of object
int attr;     // attributes of object
float avg_radius; // average radius of object used for collision detection
float max_radius; // maximum radius of object

POINT4D world_pos; // position of object in world

VECTOR4D dir;    // rotation angles of object in local
          // cords or unit direction vector user defined???

VECTOR4D ux,uy,uz; // local axes to track full orientation
          // this is updated automatically during
          // rotation calls

int num_vertices;  // number of vertices of this object
```


Playrix Ex. 1023, page 650

10 0672318350 CH07 05•02•2003 2:42 PM Page 626


```
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES]; // array of local vertices
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES]; // array of transformed vertices

int num_polys;    // number of polygons in object mesh
POLY4DV1 plist[OBJECT4DV1_MAX_POLYS]; // array of polygons

} OBJECT4DV1, *OBJECT4DV1_PTR;
```

Nothing new, but a couple `#defines` are used to size the arrays. I've set them to reasonable values:

```
// change these if you need more
#define OBJECT4DV1_MAX_VERTICES      64
#define OBJECT4DV1_MAX_POLYS        128
```

NOTE

It would be more efficient to use arrays of pointers to polygons and vertices, but I'm not worried about memory at this point. Using dynamic arrays increases the complexity of the algorithms.

Continuing with our analysis of Figure 7.1, we see that there's a simplified 3D pipeline that takes either an object or a polygon and transforms it from local coordinates all the way to screen coordinates. Let's stop here for a moment and talk about what exactly we're going to process with the 3D engine.

In most cases, a 3D engine will have a number of objects and or individual polygons that represent the geometry. At some point, all these objects and polygons will be converted into polygons and placed into a render or rasterization list. However, with simpler 3D engines, it's quite possible to keep all objects separate (and intact) and to process and render them completely as individual entities. In most cases, you don't want high-level hierarchical objects to deal with at render time. Most 3D rasterizers like polygons, not "objects," but there's nothing wrong with it if you can make it work. In fact, at first, you may think that it might be impossible to deal with objects at raster time because you don't have a single list of polygons that represent all polygons that make up all geometry in the scene. Thus, how would you sort the polygons or Z-buffer them for correct rendering order (of course, I'm getting ahead of myself here).

Well, the key is that as long as you sort the objects themselves and then sort the polygons in the objects, you could theoretically render each object itself without problems because each object defines a *convex space* (roughly speaking, it can be thought of as a bounding sphere or box for sorting purposes). Of course, if you have a Z-buffer (which we'll discuss later), it's irrelevant if you have a polygon list or a list of objects made of polygons because at some point you're going to rasterize everything and use the Z-buffer for occlusion testing on a pixel-by-pixel basis.

Playrix Ex. 1023, page 651

10 0672318350 CH07 05•02•2003 2:42 PM Page 627

The idea of the preceding paragraph is to reaffirm that we're going to do things in more than one way so that you can see the pros and cons of different techniques. Thus, I've written enough code to take a single polygon and rasterize it, or take a set of object(s) and keep them as objects all the way to rasterization, or to break objects into polygons and then add them to a master polygon list, which is what I want to talk about now.

The Master Polygon List

In most 3D engines (software or hardware), all the objects in the game are at some point converted into polygons from their high-level representations. These polygons are inserted into a master polygon list or stream, which is then pushed through the 3D pipeline (the high-level representation could be 3D meshes, parametric descriptions, or algorithmically generated on the fly). The list may be generated at the world-coordinate level or farther down the pipeline—maybe all objects are converted into polygons after the camera transformation? But, at some point, a single list of polygons will be thrown at the rasterizer. In our 3D engine toolkit, we have such a list, of course, and it's shown in Figure 7.2. If we like, we can use this list to hold the polygons. The data structure we're going to use for the first version of the polygon list is nothing more than an array of pointers to a static array, as shown in Figure 7.2 with the actual definitions following the figure.

```
typedef struct RENDERLIST4DV1_TYP
{
int state:  // state of renderlist
int attr:  // attributes of renderlist

// the render list is an array of pointers each pointing to
// a self contained "renderable" polygon face POLYF4DV1
POLYF4DV1_PTR poly_ptrs [RENDERLIST4DV1_MAX_POLYS];

// actual polygon data
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys;  // number of polys in render list

} RENDERLIST4DV1,  *RENDERLIST4DV1_PTR;
```

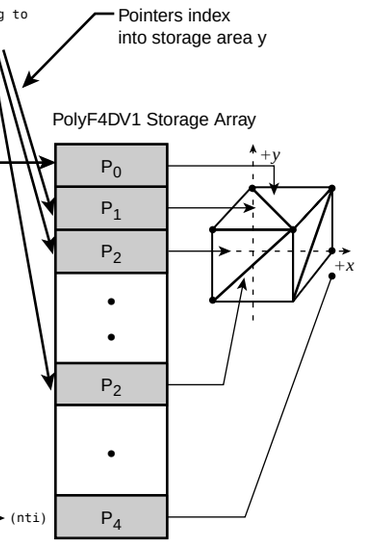


FIGURE 7.2 The master polygon list.

Playrix Ex. 1023, page 652

10 0672318350 CH07 05•02•2003 2:42 PM Page 628


```
// object to hold the render list, this way we can have more
// than one render list at a time
typedef struct RENDERLIST4DV1_TYP
{
int state; // state of renderlist ???
int attr; // attributes of renderlist ???

// the render list is an array of pointers each pointing to
// a self contained "renderable" polygon face POLYF4DV1
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

// additionally to cut down on allocatation, de-allocation
// of polygons each frame, here's where the actual polygon
// faces will be stored
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys; // number of polys in render list

} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;
```

As you can see, the `RENDERLIST4DV1` object has `state` and `attr` fields, which are used to track the state and attributes of the list (whatever we make that mean), but the important parts of the structure are the `poly_ptrs[]` and `poly_data[]` arrays. These arrays make up what's called an *indirection list* or an *indexed list*, meaning that the data is stored in the `poly_data[]` array, but instead of traversing the `poly_data[]` array directly, we traverse the `poly_ptrs[]` array, which has pointers (or it could as well have been indices into `poly_data[]`).

NOTE

The preceding technique is what Direct3D calls *indexed vertex buffers*.

Although you may think this is weird, we need it for a number of reasons. First, if I want to sort the polygons on some key—say, their z-values (for rendering order)—I can sort only the `poly_ptrs[]` list and not have to move around the `poly_data[]`! Second, if I want to disable a polygon node in the `poly_data[]` list, I would normally have to delete it and then move all the other nodes, if that was the only list I had. But with the indirection pointers in `poly_ptrs[]`, I can simply kill the link (and move the list up one if I want) to turn off a polygon node in the `poly_data[]` array. Finally, you might be thinking, "Why all this drama? Just use a linked list or something." I agree completely, but for now, a static array is easier to deal with, and only when we know that the number of polygons will vary from 2 to 2 million will using a linked list help us. For now, I know I'm going to be dealing with a few hundred polygons per frame maximum, so I can statically allocate the storage and not worry about it.

Playrix Ex. 1023, page 653

10 0672318350 CH07 05•02•2003 2:42 PM Page 629

Finally, because you might have multiple render lists with many cameras, I decided not to use globals for the render list (or the camera, for that matter), so to define a render list you would do something like this:

```
RENDERLIST4DV1 rend_list;
```

Then you would make a call to the reset function, which resets the list for the next frame of animation:

```
void Reset_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list)
{
// this function intializes and resets the sent render list and
// ready's it for polygons/faces to be inserted into it
// note that the render list in this version is composed
// of an array FACE4DV1 pointer objects, you call this
// function each frame

// since we are tracking the number of polys in the
// list via num_polys we can set it to 0
// but later we will want a more robust scheme if
// we generalize the linked list more and disconnect
// it from the polygon pointer list
rend_list->num_polys = 0; // that was hard!

} // Reset_RENDERLIST4DV1
```

Later, when we create more dynamic render lists (if we need them), this function would of course be more complex because it would probably deallocate memory and do cleanup of the last frame's polygon data.

NOTE

Data structures are so important in 3D graphics that I can't emphasize enough knowing your way around linked lists, double-linked lists, trees, graphs, and the various algorithms to search, insert, delete, and work with them. In *Tricks*, I covered a number of these data structures, but for a more complete treatise, try *Algorithms in C++*, by Sedgewick, or *Programs and Data Structures in C*, by Leendert Ammeraal.

Now that you're acquainted with the data structures, we're ready to start building a 3D wireframe engine. The funny thing is, there's not much to it, really (just a lot of details and connections, but nothing mind-boggling like filling out tax forms or anything). All we need to do is create a polygon or load an object (we'll need a function to do that) and then follow these steps:

1. Perform the local-to-world transform
2. Object removal (optional)

Playrix Ex. 1023, page 654

10 0672318350 CH07 05•02•2003 2:42 PM Page 630

3. Perform back-face removal (optional)
4. Perform the world-to-camera transform
5. Perform 3D clipping (optional)
6. Perform the camera-to-perspective transform; that is, projection
7. Perform the perspective-to-view port or screen transform
8. Rasterize polygons

NOTE

Steps 6 and 7 are sometimes merged.

Of course, some of the steps may be rearranged, and it also matters whether we're dealing with objects or converting them to polygons (we'll have to perform the conversion at some time during the pipeline). Finally, we can opt to perform the transformations using matrices, manual methods, or a hybrid. For example, it's a waste of multiplications to perform a local-to-world transform with matrices because the same can be achieved with literally three additions per point rather than a matrix multiplications with 16 multiplies and 12 additions. On the other hand, the camera transform is almost always performed with a matrix transform because the camera matrix is non-sparse. The point is that there's no correct way to do anything—if it works, and it's fast, and it works for you, that's the correct way! But in this chapter, we'll do everything in a couple of different ways to get a good grip on the various methods to apply the pipeline and how ugly some can be—especially because we're using homogeneous coordinates, which can be a good and bad thing at the same time!

The New Software Modules

The new library modules for this chapter that contain all the functional code for the 3D wireframe engine are as follows:

- `T3DLIB5.H`—The header file
- `T3DLIB5.CPP`—The C/C++ source file

Of course, to compile any program in this chapter, you'll need to include all the previous `T3DLIB*.CPP¦H` files along with all the DirectX libraries, and set the compiler to Win32 .EXE target, but you knew that, right? Additionally, I always put a comment or two at the top of each demo program listing the requisite files to compile, so if you're unsure about the necessity of any particular library file, you can always refer to the comment.

Playrix Ex. 1023, page 655

10 0672318350 CH07 05•02•2003 2:42 PM Page 631

CAUTION

Today I received about 13 emails about linker `error main() not found`. Unless I tell you we're making a DOS console application for some special reason, ALL Windows Win32/DirectX applications are Win32 .EXE and have `WinMain()` as the entry point. Thus, the only way you can get a linker error of `_main() not found` is if you're linking a Windows application as a DOS CONSOLE application and have the compiler set for CONSOLE application. I'm going to start a focus group called CAGA—Compilers Abuse Group of America—for abusers of various compiler products. If you're interested, email me at `compilerabuser@xgames3d.com` <GRIN>.

Writing a 3D File Loader

In the previous chapter, we covered a number of 3D file formats and hopefully you tried mucking with one of the 3D modelers on the CD or one you already own, can export a simple object such as a cube, and review the data in an ASCII editor. Now, writing a 3D file reader can be a simple task if you just finished a computer language course on compiler and parser design (Dragon book). Otherwise, writing a language parser might not be a nice experience for you because it's really not something you can hack together. You need to know what you're doing! So, one thing to consider is that if you're going to do a lot of 3D work, you might want to get yourself a set of 3D file import/export/conversion functions so that you don't have to write them. Nevertheless, I like doing everything myself, so I'm going to write every single format reader we use in this book rather than use third-party stuff. I'll try to make them short, clear, and to the point.

With that in mind, we're probably going to use the .PLG file format and the .NFF formats first in this book because they're simple to understand, and I can write a parser for either of them before Sci-Fi Channel's *First Wave* comes on tonight, and that's an important consideration. Moreover, they're both English-readable formats, and you can write the files by hand, which is what we're going to do for this chapter. Why? I want you to be able to work with 3D coordinates in your head before you delegate it to a 3D modeler.

We'll start with a .PLG parser, but I'm not going to go over the entire .PLG file format in detail because it's in the last chapter. I'm just going to give you a generic description of what a file we are trying to read looks like. Remember that the format is basically a vertex list–based format with a list of polygons referring to the vertex list as indices. Here's what a generic .PLG file looks like:

```
#this is a comment
#header line object name, number of vertices and polygons
object_name num_vertices num_polygons

# vertex list
# each vertex is in the form of x y z
x0 y0 z0
```


Playrix Ex. 1023, page 656

10 0672318350 CH07 05•02•2003 2:42 PM Page 632


```
x1 y1 z1
x2 y2 z2
x3 y3 z3
.
.
xn yn zn

# polygon list
# each polygon is in the form
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
.
.
surface_descriptor num_vertices v0 v1 . vn
#where v0 v1 . vn are the vertex numbers that make up each polygon
```

And here's a simple cube object to give you something to look at for reference:

```
# begin plg/plx file

# simple cube, 8 vertices, 12 polygons
tri 8 12

# vertex list in x y z format
 5 5 5
-5 5 5
-5 5 -5
 5 5 -5
 5 -5 5
-5 -5 5
-5 -5 -5
 5 -5 -5

# polygon list
# each poly has 3 vertices
0xd0f0 3 0 1 2
0xd0f0 3 0 2 3
0xd0f0 3 0 7 4
0xd0f0 3 0 3 7
0xd0f0 3 4 7 6
0xd0f0 3 4 6 5
0xd0f0 3 1 6 2
```

Playrix Ex. 1023, page 657

10 0672318350 CH07 05•02•2003 2:42 PM Page 633


```
0xd0f0 3 1 5 6
0xd0f0 3 3 6 7
0xd0f0 3 3 2 6
0xd0f0 3 0 4 5
0xd0f0 3 0 5 1
```

The only thing about PLG files is they are a bit "80s," so I've altered them a bit by adding some functionality; I call the new ones .PLX files. The point is that if you do actually find a 3D modeler that exports .PLG files, the reader we'll write will still read them, but the 16-bit surface descriptor will be invalid. The new surface descriptor I've made up for the modified .PLG files (which I sometimes refer to as .PLX files) is defined here:

Bit Number: `d15` `d0`

Encoding: `CSSD ¦ RRRR¦ GGGG ¦ BBBB`

where

- `C` is the RGB/indexed color flag.
- `SS` are two bits that define the shading mode.
- `D` is the double-sided flag.
- `RRRR`, `GGGG`, `BBBB` are the red, green, and blue bits for RGB mode.
- `GGGGBBBB` is the 8-bit color index for 8-bit mode and interpreted as the color index.

The following `#defines` help discriminate and extract these bit fields:

```
// bit masks to simplify testing????
#define PLX_RGB_MASK    0x8000// mask extracts RGB/indexed color
#define PLX_SHADE_MODE_MASK 0x6000// mask extracts shading mode
#define PLX_2SIDED_MASK   0x1000// mask for double sided
#define PLX_COLOR_MASK   0x0fff// xxxxrrrrggggbbbb, 4-bits per channel RGB
                   // xxxxxxxxiiiiiiii, indexed mode 8-bit index

// these are the comparison flags after masking
// color mode of polygon
#define PLX_COLOR_MODE_RGB_FLAG   0x8000 // poly uses RGB color
#define PLX_COLOR_MODE_INDEXED_FLAG 0x0000 // poly uses indexed 8-bit color

// double sided flag
#define PLX_2SIDED_FLAG      0x1000 // poly is double sided
#define PLX_1SIDED_FLAG      0x0000 // poly is single sided
```


Playrix Ex. 1023, page 658

10 0672318350 CH07 05•02•2003 2:42 PM Page 634


```
// shading mode of polygon
#define PLX_SHADE_MODE_PURE_FLAG  0x0000 // poly is a constant color
#define PLX_SHADE_MODE_FLAT_FLAG  0x2000 // poly uses flat shading
#define PLX_SHADE_MODE_GOURAUD_FLAG 0x4000 // poly used gouraud shading
#define PLX_SHADE_MODE_PHONG_FLAG  0x6000 // poly uses phong shading
```

For example, if you want to define a polygon that's two-sided, 16-bit RGB color (0x3,0x8,0xF), with Gouraud shading, you would build the following surface descriptor.

Given that the 16-bit surface descriptor is in the format:

`CSSD ¦ RRRR¦ GGGG ¦ BBBB`

and

`C` = 1 for 16-bit color

`SS` = 10 for Gouraud shading

`D` = Double-sided bit; 1 for double sided

The `CCSD` nibble would look like this:

`1101`

The `RRRR ¦ GGGG ¦ BBBB` nibbles would simple be 0x3, 0x8, and 0xf. Thus, the result is

`1101 0011 1000 1111 = 0xD38F`

And that's all there is to it. The upper four bits of each descriptor describe the properties of the polygon and the color model, and the remaining 12 bits describe the color in the RGB model, or the index in the 8-bit model. All the file reader has to do is break these bits apart and set the appropriate bits in the polygon definition of whatever data structure(s) we load the object into. So, let's see the functions that read in a .PLG file.

The .PLG/X File Loader

The good thing about .PLG files is that their format is fixed, meaning that you can't put the polygons and then the vertices; you must put the vertices and then the polygons (.DXF files, for example, are a little more free form). Each data line in a .PLG must be in the correct format, which is usually one or more numbers separated by whitespace. So, the plan of attack is to write a simple parser that reads in each line along with a crude finite state machine that knows that the .PLG format is broken up into three sections:

Section 1: Header info

Section 2: Vertex data info

Section 3: Polygon data info

Playrix Ex. 1023, page 659

10 0672318350 CH07 05•02•2003 2:42 PM Page 635

Each section has a particular format and is *line-based data*. Meaning we're guaranteed that each data line doesn't continue on to multiple lines. Thus, we can write a parser that reads single lines and then, depending on the state of the parser (what section we are in), we can parse the lines and extract the data elements (which are separated by whitespace). That's all there is to it. Of course, as the parser is parsing the data, the results are inserted into some data structure (which in this case will be an `OBJECT4DV1` object).

To write the parser, I could use a single function, but it would be pretty ugly with all the various cases and the line reader/tokenizer. So, I decided to break it up into two functions which works pretty well.

TIP

If I did the parsing the right way, I would first write a suite of parsing functions and then leverage them to write this reader using a language specification. Of course, there are commercial tools to do this, including YACC and LEX, which are both Unix-based tools (but, of course, are available on PC) to write compilers and interpreters.

The first support function, `Get_Line_PLG()`, reads a single line of the .PLG file (it's based on `fgets()`) and is shown here:

```
char *Get_Line_PLG(char *buffer, int maxlength, FILE *fp)
{
// this little helper function simply reads past comments
// and blank lines in a PLG file and always returns full
// lines with something on them on NULL if the file is empty

int index = 0; // general index
int length = 0; // general length

// enter into parsing loop
while(1)
   {
   // read the next line
   if (!fgets(buffer, maxlength, fp))
   return(NULL);

  // kill the whitespace
  for (length = strlen(buffer), index = 0;
     isspace(buffer[index]); index++);

  // test if this was a blank line or a comment
  if (index >= length || buffer[index]=='#')
    continue;
```


Playrix Ex. 1023, page 660

10 0672318350 CH07 05•02•2003 2:42 PM Page 636


```
  // at this point we have a good line
  return(&buffer[index]);
  } // end while

} // end Get_Line_PLG
```

There's not much to it. It basically understands whitespace and comments delineated by the # symbol. The function simply returns the next line in the file or *NULL* if the file is empty, which would occur only in an error condition because the .PLG file will have an exact deterministic number of data tokens based on the header information line. The function that reads the actual .PLG file is called `Load_OBJECT4DV1_PLG()` (note the naming convention) and is shown next:

```
int Load_OBJECT4DV1_PLG(OBJECT4DV1_PTR obj, // pointer to object
          char *filename,   // filename of plg file
          VECTOR4D_PTR scale,   // initial scaling factors
          VECTOR4D_PTR pos,    // initial position
          VECTOR4D_PTR rot)    // initial rotations
{
// this function loads a plg object in off disk, additionally
// it allows the caller to scale, position, and rotate the object
// to save extra calls later for non-dynamic objects

FILE *fp;     // file pointer
char buffer[256]; // working buffer

char *token_string; // pointer to actual token text, ready for parsing

// file format review, note types at end of each description
// # this is a comment

// # object descriptor
// object_name_string num_verts_int num_polys_int

// # vertex list
// x0_float y0_float z0_float
// x1_float y1_float z1_float
// x2_float y2_float z2_float
// .
// .
// xn_float yn_float zn_float
//
// # polygon list
// surface_description_ushort num_verts_int v0_index_int v1_index_int .
```

Playrix Ex. 1023, page 661

10 0672318350 CH07 05•02•2003 2:42 PM Page 637


```
// vn_index_int
// .
// .
// surface_description_ushort num_verts_int v0_index_int v1_index_int .
// vn_index_int

// lets keep it simple and assume one element per line
// hence we have to find the object descriptor, read it in, then the
// vertex list and read it in, and finally the polygon list -- simple :)

// Step 1: clear out the object and initialize it a bit
memset(obj, 0, sizeof(OBJECT4DV1));

// set state of object to active and visible
obj->state = OBJECT4DV1_STATE_ACTIVE ¦ OBJECT4DV1_STATE_VISIBLE;

// set position of object
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

// Step 2: open the file for reading
if (!(fp = fopen(filename, "r")))
  {
  printf("Couldn't open PLG file %s.", filename);
  return(0);
  } // end if

// Step 3: get first token string which should be the object descriptor
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
  {
  Write_Error("PLG file error with file %s (object descriptor invalid).",
        filename);
  return(0);
  } // end if

Write_Error("Object Descriptor: %s", token_string);

// parse out the info object
sscanf(token_string, "%s %d %d", obj->name,
                &obj->num_vertices,
                 &obj->num_polys);
```


Playrix Ex. 1023, page 662

10 0672318350 CH07 05•02•2003 2:42 PM Page 638


```
// Step 4: load the vertex list
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // get the next vertex
  if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
     {
     Write_Error("PLG file error with file %s (vertex list invalid).",
           filename);
     return(0);
     } // end if

  // parse out vertex
  sscanf(token_string, "%f %f %f", &obj->vlist_local[vertex].x,
                  &obj->vlist_local[vertex].y,
                  &obj->vlist_local[vertex].z);
  obj->vlist_local[vertex].w = 1;

  // scale vertices
  obj->vlist_local[vertex].x*=scale->x;
  obj->vlist_local[vertex].y*=scale->y;
  obj->vlist_local[vertex].z*=scale->z;

  Write_Error("\nVertex %d = %f, %f, %f, %f", vertex,
                      obj->vlist_local[vertex].x,
                      obj->vlist_local[vertex].y,
                      obj->vlist_local[vertex].z,
                      obj->vlist_local[vertex].w);

  } // end for vertex

// compute average and max radius
Compute_OBJECT4DV1_Radius(obj);

Write_Error("\nObject average radius = %f, max radius = %f",
      obj->avg_radius, obj->max_radius);

int poly_surface_desc = 0; // PLG/PLX surface descriptor
int poly_num_verts  = 0; // number of vertices for currpoly (always 3)
char tmp_string[8];    // string to hold surface descriptor in and
              // test if it need to be converted from hex

// Step 5: load the polygon list
for (int poly=0; poly < obj->num_polys; poly++)
  {
```

Playrix Ex. 1023, page 663

10 0672318350 CH07 05•02•2003 2:42 PM Page 639


```
// get the next polygon descriptor
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
  {
  Write_Error("PLG file error with file %s (polygon descriptor invalid).",
         filename);
  return(0);
  } // end if

Write_Error("\nPolygon %d:", poly);

// each vertex list MUST have 3 vertices since we made this a rule
// that all models
// must be constructed of triangles
// read in surface descriptor, number of vertices, and vertex list
sscanf(token_string, "%s %d %d %d %d", tmp_string,
                &poly_num_verts, // should always be 3
                &obj->plist[poly].vert[0],
                &obj->plist[poly].vert[1],
                &obj->plist[poly].vert[2]);


// since we are allowing the surface descriptor to be in hex format
// with a leading "0x" we need to test for it
if (tmp_string[0] == '0' && toupper(tmp_string[1]) == 'X')
  sscanf(tmp_string,"%x", &poly_surface_desc);
else
  poly_surface_desc = atoi(tmp_string);

// point polygon vertex list to object's vertex list
// note this is redundant since the polylist is contained
// within the object and its up to the user to select
// whether the local or transformed vertex list is used when building
// polygon geometry, might be better to set to NULL in the context
// of polygons that are part of an object
obj->plist[poly].vlist = obj->vlist_local;

Write_Error("\nSurface Desc = 0x%.4x, num_verts = %d, /
       vert_indices [%d, %d, %d]",
                      poly_surface_desc,
                      poly_num_verts,
                      obj->plist[poly].vert[0],
                      obj->plist[poly].vert[1],
                      obj->plist[poly].vert[2]);
```


Playrix Ex. 1023, page 664

10 0672318350 CH07 05•02•2003 2:42 PM Page 640


```
// now we that we have the vertex list and we have entered the polygon
// vertex index data into the polygon itself, now let's analyze the
// surface descriptor and set the fields for the polygon based on
// the description

// extract out each field of data from the surface descriptor
// first let's get the single/double sided stuff out of the way
if ((poly_surface_desc & PLX_2SIDED_FLAG))
  {
  SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_2SIDED);
  Write_Error("\n2 sided.");
  } // end if
else
  {
  // one sided
  Write_Error("\n1 sided.");
  } // end else

// now let's set the color type and color
if ((poly_surface_desc & PLX_COLOR_MODE_RGB_FLAG))
  {
  // this is an RGB 4.4.4 surface, I know sucks!
  SET_BIT(obj->plist[poly].attr,POLY4DV1_ATTR_RGB16);

  // now extract color and copy into polygon color
  // field in proper 16-bit format
  // 0x0RGB is the format, 4 bits per pixel
  int red  = ((poly_surface_desc & 0x0f00) >> 8);
  int green = ((poly_surface_desc & 0x00f0) >> 4);
  int blue = (poly_surface_desc & 0x000f);

  // although the data is always in 4.4.4 format, the graphics card
  // is 5.5.5 or 5.6.5, but our virtual color system translates
  // 8.8.8 into 5.5.5 or 5.6.5 for us, but we have to first scale all
  // these 4.4.4 values into 8.8.8
  obj->plist[poly].color = RGB16Bit(red*16, green*16, blue*16);
  Write_Error("\nRGB color = [%d, %d, %d]", red, green, blue);
  } // end if
else
  {
  // this is an 8-bit color indexed surface
  SET_BIT(obj->plist[poly].attr,POLY4DV1_ATTR_8BITCOLOR);
```

Playrix Ex. 1023, page 665

10 0672318350 CH07 05•02•2003 2:42 PM Page 641


```
    // and simply extract the last 8 bits and that's the color index
    obj->plist[poly].color = (poly_surface_desc & 0x00ff);

    Write_Error("\n8-bit color index = %d", obj->plist[poly].color);

    } // end else

  // handle shading mode
  int shade_mode = (poly_surface_desc & PLX_SHADE_MODE_MASK);

  // set polygon shading mode
  switch(shade_mode)
     {
     case PLX_SHADE_MODE_PURE_FLAG: {
     SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_PURE);
     Write_Error("\nShade mode = pure");
     } break;

     case PLX_SHADE_MODE_FLAT_FLAG: {
     SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_FLAT);
     Write_Error("\nShade mode = flat");

     } break;

     case PLX_SHADE_MODE_GOURAUD_FLAG: {
     SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_GOURAUD);
     Write_Error("\nShade mode = gouraud");
     } break;

     case PLX_SHADE_MODE_PHONG_FLAG: {
     SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_PHONG);
     Write_Error("\nShade mode = phong");
     } break;

     default: break;
     } // end switch

  // finally set the polygon to active
  obj->plist[poly].state = POLY4DV1_STATE_ACTIVE;

  } // end for poly

// close the file
fclose(fp);
```


Playrix Ex. 1023, page 666

10 0672318350 CH07 05•02•2003 2:42 PM Page 642


```
// return success
return(1);

} // end Load_OBJECT4DV1_PLG
```

The function has a bit of error checking, which I probably went overboard with, but I wanted you to get in the habit of adding error handling to file loading functions because they can really go awry and write into memory. (Mr. Xenon, if you just initialized your variables, that wouldn't happen to you!) As a rule, I never do error handling in game code because if the code's not perfect, I'll make it perfect. But in a loader, you're counting on an external file to be correct, which might not be the case, depending the source of the file. Thus, some nice error states are dumped out to the error file as the loader is loading the data.

Moving on to how the function works, the function opens up the .PLG file requested by the string in ***filename***. Then the first line of the file is read in, which controls the number of vertices and polygons that need to be read in. The function then methodically reads in the vertex list, inserting the vertices into the sent `OBJECT4DV1` in `obj`. It then moves on to the polygons and inserts them along with determining their flags and color models via the surface descriptor for each polygon, which are translated into `POLY_ATTR_*` flags and logically `OR`ed. Additionally, you can send some parameters to the function to transform the objects as it's loaded: `scale`, `pos`, and `rot` are the initial scaling, position, and rotation factors for the object (note that they're vectors).

The only other interesting thing about the function is that it computes the average and maximum radius of the object for collision detection and object removal later down the line in the 3D pipeline. Anyway, to load an object off disk, all you do is this:

```
OBJECT4DV1 obj; // object storage

VECTOR4D scale = {1,1,1,1}, // no scaling basically
     pos  = {0,0,0,1}, // pos (0,0,0) in world
     rot  = {0,0,0,1}; // no rotation

// load the object
Load_OBJECT4DV1_PLG(&obj, "cube.plg", &scale, &pos, &rot);
```

That's it. If you use the debugger or something to read through the data structure `obj`, you'll see that the object data has been read into it and it's all correct (I hope!). The act of loading an external file into the `OBJECT4DV1` initializes the object and sets up the `state` and `attr` fields as well as setting the initial orientation basis defined by `ux`, `uy`, `uz` to `ux` = <1,0,0,1>, `uy` = <0,1,0,1>, and `uz` = <0,0,1,1>, which is in alignment with the world coordinate system axes for reference. Now, let's recap where we are as far as geometry.

Playrix Ex. 1023, page 667

10 0672318350 CH07 05•02•2003 2:42 PM Page 643

At this point, we have three different data structures that we can work with:

- `POLY4DV1`—A single polygon, but needs an external vertex list.
- `POLYF4DV1`—A single polygon that has its own internal vertice storage. Also the basis of the master polygon list.
- `OBJECT4DV1`—A single object that's composed of a number of polygons. It has its own vertex and polygon storage; that is, it's self-contained.

We could write our engine to handle all of these types all the way down the pipeline, but I've opted to handle only the `POLYF4DV1` and the `OBJECT4DV1` because the `POLY4DV1` is just a `POLYF4DV1` without a vertex list. So, let's start building an engine to move these guys (yes…polygons are boys, but pixels are girls) through the pipeline!

Building the 3D Pipeline

We're ready to start implementing the 3D pipeline. We have our data structures, and we can load external objects in .PLG (enhanced .PLG, actually) format, so there's not much to do other than write the darn functions. Hold on one sec…let me write a couple thousand lines…okay, done!

CAUTION

The only regret I have is that we're going to have to write a lot of functions before we get to see any demos that put anything on the screen because the functions are all interrelated. But what I'll do at that point is create a number of demos that show some of the techniques leading up to the rendering stage of the pipeline.

Generic Transformation Functions

If you recall, anything that you can do in 3D, you can do via a 4x4 matrix transformation along with a post 4D homogeneous to 3D conversion (division by w, basically). However, this doesn't always mean it's the fastest way to do it! In general, any transformation that is a concatenation of rotations, translations, and projections is better to perform with a single 4x4 matrix because the matrix represents numerous transforms concantenated together and is usually non-sparse. However, if you're just trying to translate a single point, **p**, using a matrix multiple is overkill because it requires (4*4) = 16 multiplications, (4*3) = 12 additions to process the entire matrix multiple of **p***M = **p**(1x4) * **M**(4x4). And if you do the translation manually, you only need three additions:

```
x=x+xt;
y=y+yt;
z=z+zt;
w=w;
```

Playrix Ex. 1023, page 668

10 0672318350 CH07 05•02•2003 2:42 PM Page 644

The point is that in a production 3D engine, you might find a manual method faster and simpler. However, you might also find that although doing it manually is faster, it's not as clean or doesn't port well to hardware (which almost always uses 4x4 transforms for everything). Hence, as much as possible, we're going to allow both approaches in our engine design.

To facilitate the general transformation of a polygon or object by a 4x4 matrix, we need a function that takes as a parameter the polygon (`POLYF4DV1`) or the object (`OBJECT4DV1`) and performs a matrix multiplications on each vertex that makes up the geometry with a transformation matrix. Additionally, we have to take into consideration that both of these data structures have both local and transformed vertex storage.

If you recall, once you transform an object's vertex data, the transformation is lost. Thus, you better make sure that you want to do it. In many cases you do, but in some cases, the transformation is part of the pipeline, so you want to keep a copy of the original data unscathed, but transform and store the new data as it moves through the pipeline in secondary storage. Both the `POLYF4DV1` and `OBJECT4DV1` have such a secondary transformed storage space.

In the case of `POLYF4DV1`, the vertex data in the structure is stored in

```
POINT4D vlist[3]; // the vertices of this triangle

POINT4D tvlist[3]; // the vertices after transformation if needed
```

where `tvlist[]` is used to hold the transformed data. Similarly, in the `OJBECT4DV1` structure, vertex data is stored in these arrays:

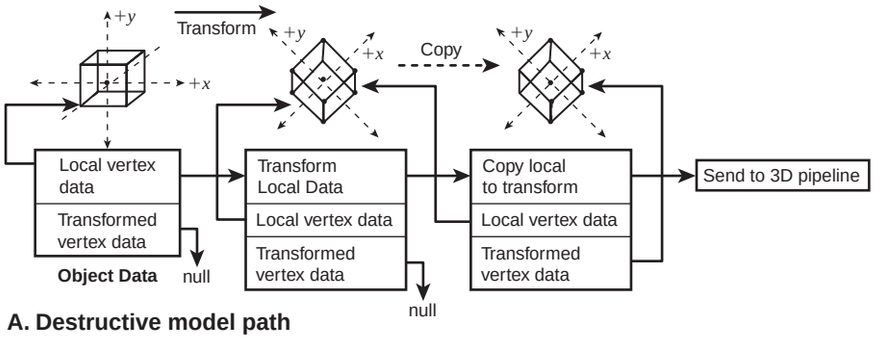
```
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES]; // array of local vertices

POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES]; // array of transformed vertices
```

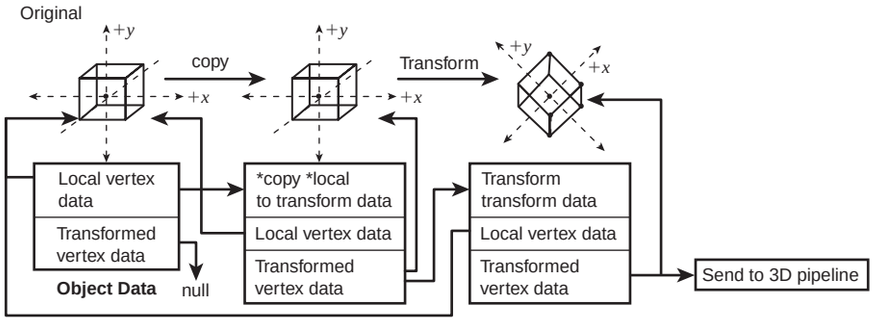
Hence, you can obliterate the secondary storage as much as you like because you always have a copy of the original data in the local storage array. However, sometimes you want to transform the local storage data (a scale or rotation operation, perhaps for animation) and want the new local model to remember the transform. In that case, you would modify the local data itself. Figure 7.3 shows a flowchart of the two operations we want to perform on vertex data.

Playrix Ex. 1023, page 669

10 0672318350 CH07 05•02•2003 2:42 PM Page 645



A. Destructive model path



B. Non-destuctive model path

FIGURE 7.3 Flowchart of destructive and non-destructive transformations on vertex data.

So, we need a function to transform a `POLYF4DV1` by a transformation matrix. But, in most cases, you wouldn't want to transform a single polygon (it's a waste). You would wait until you had an entire list of them, so let's think in terms of transforming a list of polygons, such as the master polygon list object, which we called a `RENDERLIST4DV1` and is nothing more than an array of `POLYF4DV1`s. Here's a function that performs the transform defined by the matrix `mt` to a render list:

```
void Transform_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
        MATRIX4X4_PTR mt,  // transformation matrix
        int coord_select)  // selects coords to transform
{
// this function simply transforms all of the
// polygons vertices in the local or trans
// array of the render list by the sent matrix

// what coordinates should be transformed?
switch(coord_select)
{
```

Playrix Ex. 1023, page 670

10 0672318350 CH07 05•02•2003 2:42 PM Page 646


```
case TRANSFORM_LOCAL_ONLY:
{
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // acquire current polygon
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // is this polygon valid?
  // transform polygon if and only if it's not clipped, not culled,
  // active, and visible, note however the concept of "backface" is
  // irrelevant in a wire frame engine though
  if ((curr_poly==NULL) ¦¦
       !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

  // all good, let's transform
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // transform the vertex by mt
    POINT4D presult; // hold result of each transformation

    // transform point
    Mat_Mul_VECTOR4D_4X4(&curr_poly->vlist[vertex], mt, &presult);

    // store result back
    VECTOR4D_COPY(&curr_poly->vlist[vertex], &presult);
    } // end for vertex
  } // end for poly

} break;

case TRANSFORM_TRANS_ONLY:
{
// transform each "transformed" vertex of the render list
// remember, the idea of the tvlist[] array is to accumulate
// transformations
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // acquire current polygon
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];
  // is this polygon valid?
```

Playrix Ex. 1023, page 671

10 0672318350 CH07 05•02•2003 2:42 PM Page 647


```
  // transform this polygon if and only if it's not clipped, not culled,
  // active, and visible, note however the concept of "backface" is
  // irrelevant in a wire frame engine though
  if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

  // all good, let's transform
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // transform the vertex by mt
    POINT4D presult; // hold result of each transformation

    // transform point
    Mat_Mul_VECTOR4D_4X4(&curr_poly->tvlist[vertex], mt, &presult);
    // store result back
    VECTOR4D_COPY(&curr_poly->tvlist[vertex], &presult);
    } // end for vertex
  } // end for poly

} break;

case TRANSFORM_LOCAL_TO_TRANS:
{
// transform each local/model vertex of the render list and store result
// in "transformed" vertex list
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // acquire current polygon
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // is this polygon valid?
  // transform polygon if and only if it's not clipped, not culled,
  // active, and visible, note however the concept of "backface" is
  // irrelevant in a wire frame engine though
  if ((curr_poly==NULL) ¦¦
       !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly
```


Playrix Ex. 1023, page 672

10 0672318350 CH07 05•02•2003 2:42 PM Page 648


```
  // all good, let's transform
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // transform the vertex by mt
    Mat_Mul_VECTOR4D_4X4(&curr_poly->vlist[vertex], mt,
                  &curr_poly->tvlist[vertex]);
    } // end for vertex

  } // end for poly

} break;

default: break;

} // end switch

} // end Transform_RENDERLIST4DV1
```

The function is fairly straightforward. It simply applies the 4x4 transformation matrix `mt` to `rend_list` based on the coordinate transformation selection flag `coord_select`. Then, depending on the requested coordinates that should be transformed, the function starts looping and making calls to the matrix multiply function. The coordinate transformation selector is controlled by the parameter `coord_select` based on the following defines:

```
// transformation control flags
#define TRANSFORM_LOCAL_ONLY   0 // perform the transformation
                   // in place on the
                   // local/world vertex list
#define TRANSFORM_TRANS_ONLY   1 // perform the transformation
                   // in place on the
                   // "transformed" vertex list

#define TRANSFORM_LOCAL_TO_TRANS 2 // perform the transformation
                   // to the local
                   // vertex list, but store the
                   // results in the
                   // transformed vertex list
```

Thus, you can apply the transform to (1) the local storage, (2) the transformed storage, or (3) perform the transform to the local storage and store the results in the transformed storage. Did I leave anything out?

Using the function is trivial; assume that we have a render list named `rend_list` full of polygons and we want to transform the local coordinates of each polygon within it with

Playrix Ex. 1023, page 673

10 0672318350 CH07 05•02•2003 2:42 PM Page 649

the matrix m_trans and store the results in the transformed storage. The following call would do the job:

```
Transform_RENDERLIST4DV1(&rend_list, m_trans, TRANSFORM_LOCAL_TO_TRANS );
```

Wham, bam, the entire list is transformed with the 4x4 matrix. Cool, huh?

The second function we need is one to transform an object itself or the OBJECT4DV1. Basically, the function will be identical, but a little simpler because there's just a vertex list rather than a collection of polygons, each with its own vertex list. Of course, we still want the same coordinate transformation selection capability. With that in mind, here's the function:

```
void Transform_OBJECT4DV1(OBJECT4DV1_PTR obj, // object to transform
         MATRIX4X4_PTR mt,  // transformation matrix
         int coord_select,  // selects coords to transform
         int transform_basis) // flags if vector orientation
                    // should be transformed to
{
// this function simply transforms all of the vertices
// in the local or trans
// array by the sent matrix

// what coordinates should be transformed?
switch(coord_select)
   {
   case TRANSFORM_LOCAL_ONLY:
   {
   // transform each local/model vertex of the object mesh in place
   for (int vertex=0; vertex < obj->num_vertices; vertex++)
     {
     POINT4D presult; // hold result of each transformation

     // transform point
     Mat_Mul_VECTOR4D_4X4(&obj->vlist_local[vertex], mt, &presult);

     // store result back
     VECTOR4D_COPY(&obj->vlist_local[vertex], &presult);
     } // end for index
   } break;

   case TRANSFORM_TRANS_ONLY:
   {
   // transform each "transformed" vertex of the object mesh in place
   // remember, the idea of the vlist_trans[] array is to accumulate
```

Playrix Ex. 1023, page 674

10 0672318350 CH07 05•02•2003 2:42 PM Page 650


```
    // transformations
    for (int vertex=0; vertex < obj->num_vertices; vertex++)
      {
      POINT4D presult; // hold result of each transformation

      // transform point
      Mat_Mul_VECTOR4D_4X4(&obj->vlist_trans[vertex], mt, &presult);

      // store result back
      VECTOR4D_COPY(&obj->vlist_trans[vertex], &presult);
      } // end for index

    } break;

    case TRANSFORM_LOCAL_TO_TRANS:
    {
    // transform each local/model vertex of the object mesh
    // and store result
    // in "transformed" vertex list
    for (int vertex=0; vertex < obj->num_vertices; vertex++)
      {
      POINT4D presult; // hold result of each transformation

      // transform point
      Mat_Mul_VECTOR4D_4X4(&obj->vlist_local[vertex], mt,
      &obj->vlist_trans[vertex]);

      } // end for index
    } break;

    default: break;

} // end switch

// finally, test if transform should be applied to orientation basis
// hopefully this is a rotation, otherwise the basis will get corrupted
if (transform_basis)
  {
  // now rotate orientation basis for object
  VECTOR4D vresult; // use to rotate each orientation vector axis

  // rotate ux of basis
  Mat_Mul_VECTOR4D_4X4(&obj->ux, mt, &vresult);
  VECTOR4D_COPY(&obj->ux, &vresult);
```

Playrix Ex. 1023, page 675

10 0672318350 CH07 05•02•2003 2:42 PM Page 651


```
  // rotate uy of basis
  Mat_Mul_VECTOR4D_4X4(&obj->uy, mt, &vresult);
  VECTOR4D_COPY(&obj->uy, &vresult);

  // rotate uz of basis
  Mat_Mul_VECTOR4D_4X4(&obj->uz, mt, &vresult);
  VECTOR4D_COPY(&obj->uz, &vresult);
  } // end if

} // end Transform_OBJECT4DV1
```

Again, nothing more than the three different cases with the matrix multiplication math invoked, but there's something else here. Aha! There's a `transform_basis` flag. This is needed for the following reason: As shown in Figure 7.4, when you transform a 3D object, you lose its original load time orientation. That is, if you know you load an object with "up" pointing in the +Y direction and "forward" in the +Z direction, how will you track the result orientation after transforming an object?

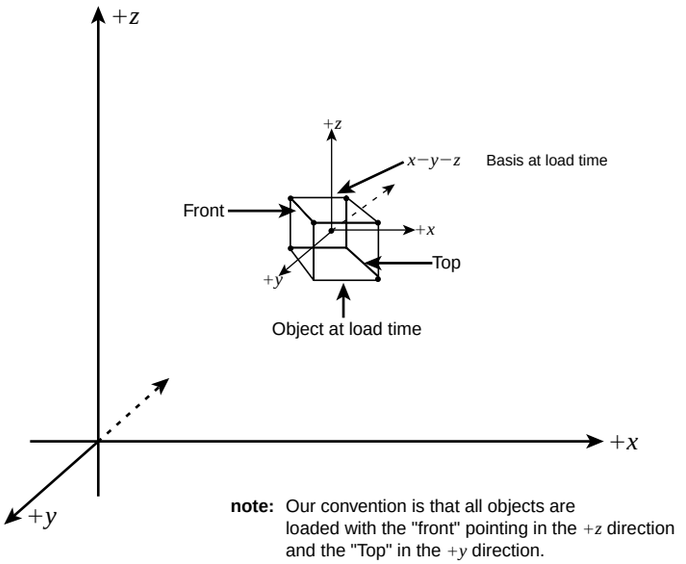


FIGURE 7.4 Load time orientation of an object.

The point is that you need to also transform the basis or local axes or the object, too. Then you can always use the basis of the object to figure the orientation of the object by comparing the basis to the world axes (a very common problem that I get billions of emails on…okay, maybe only hundreds of millions of emails). Figure 7.5 shows the same

Playrix Ex. 1023, page 676

10 0672318350 CH07 05•02•2003 2:42 PM Page 652

object after transformation with the object's local basis transformed in addition to the vertex data. Thus, the `transform_basis` flag determines whether you want the function also to transform the basis of the object itself. Remember that the basis of the object could be anything to start with, but usually it's [ux = <1,0,0,0>, uy = <0,1,0,0>, uz = <0,0,1,0>], which is the world unit coordinate axes basis.

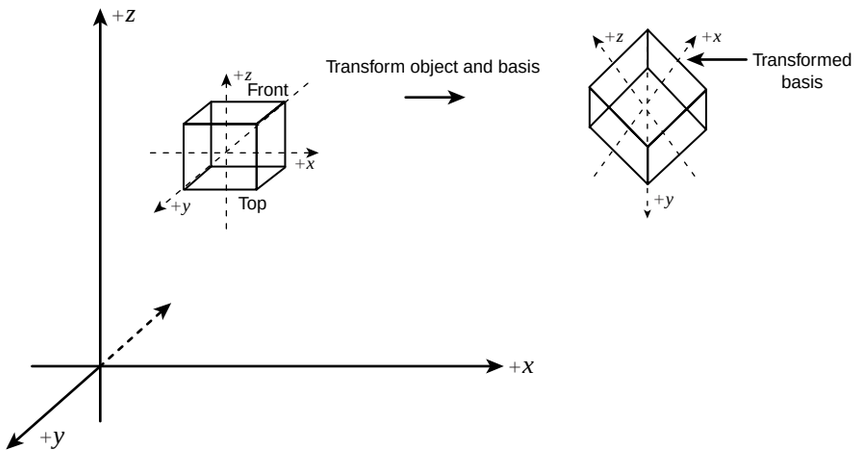


FIGURE 7.5 Local basis transformation to track an object's orientation.

As an example, say that we had previously loaded an object into the variable `obj` to transform its local coordinates by the matrix `m_trans` along with its basis. Here's the call:

```
Transform_OBJECTDV1(&obj, m_trans, TRANSFORM_LOCAL_TO_TRANS, 1 );
```

Easy as cheating on taxes! But of course, I would never do that.

Now that we have a general transformation function, we can implement any transformation as a 4x4 matrix and then call this function. But remember that many of the transforms are faster if performed manually. In the following sections, we'll see how to perform the various steps of the pipeline using manual and matrix methods on each geometry object: `OBJECT4DV1` and `RENDERLIST4DV1`.

The Local-to-World Transform

The local-to-world transformation transforms an object's (or polygon's) local or model coordinates into world coordinates (which are what defines the entire game universe), as shown in Figure 7.6. Therefore, given a point or vertex defined by **p** = <x,y,z,1> and a translation factor **dt** = <xt, yt, zt, 1>, the local-to-world transform is simply

```
p' = p + dt
```

leaving the w = 1 component the same.

Playrix Ex. 1023, page 677

10 0672318350 CH07 05•02•2003 2:42 PM Page 653

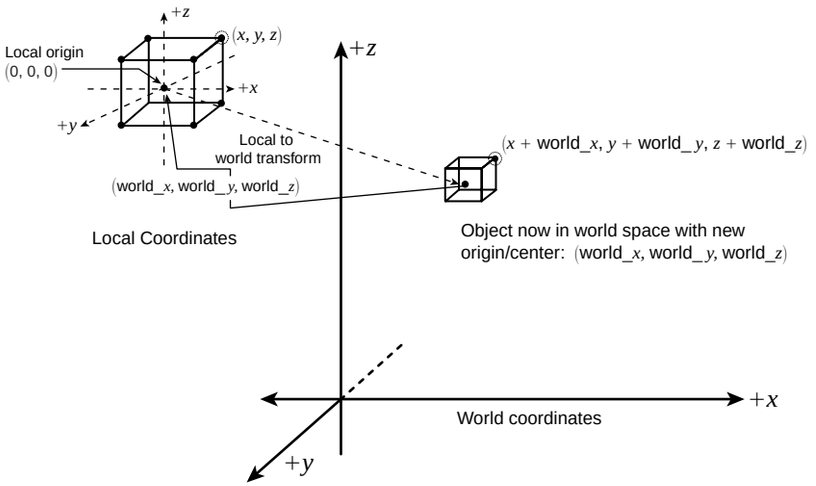


FIGURE 7.6 Local-to-world transformation.

Local-to-World Transformation for Objects

The function that performs the local (model)-to-world transformation on an `OBJECT4DV1` simply translates the vertices of the object stored in `local_vlist[]` by the amount `world_pos` and stores the results in `trans_vlist[]`. Here's the manual version:

```
void Model_To_World_OBJECT4DV1(OBJECT4DV1_PTR obj,
         int coord_select= TRANSFORM_LOCAL_TO_TRANS)
{
// NOTE: Not matrix based
// this function converts the local model coordinates of the
// sent object into world coordinates, the results are stored
// in the transformed vertex list (vlist_trans) within the object

// interate thru vertex list and transform all the model/local
// coords to world coords by translating the vertex list by
// the amount world_pos and storing the results in vlist_trans[]

if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
{
for (int vertex=0; vertex < obj->num_vertices; vertex++)
  {
  // translate vertex
  VECTOR4D_Add(&obj->vlist_local[vertex], &obj->world_pos,
         &obj->vlist_trans[vertex]);
  } // end for vertex
```


Playrix Ex. 1023, page 678

10 0672318350 CH07 05•02•2003 2:42 PM Page 654


```
} // end if local
else // TRANSFORM_TRANS_ONLY)
{
for (int vertex=0; vertex < obj->num_vertices; vertex++)
  {
  // translate vertex
  VECTOR4D_Add(&obj->vlist_trans[vertex], &obj->world_pos,
         &obj->vlist_trans[vertex]);
  } // end for vertex
} // end else trans

} // end Model_To_World_OBJECT4DV1
```

Given that you've already loaded a unit cube into the `OBJECT4DV1` cube and then set its `world_pos` to (100,200,400), you can translate the whole mesh with the call

```
Model_To_World_OBJECT4DV1(&cube);
```

Notice that there's a coordinate select function. This is useful in the cases where you perform another transformation to the model that's already in the transformed coordinates. In this case, the model to world transform should use the transformed coordinate rather than the original model coordinates. To control this, use the `coord_select` variable as usual.

Now, if you wanted to, you could perform this transformation with a matrix, but it would really be a waste of cycles. However, we need to look at it, because hardware loves matrices, and when you take advantage of it you will see this. The function that builds up translation matrix is called `void Build_Model_To_World_MATRIX4X4()` and is shown here:

```
void Build_Model_To_World_MATRIX4X4(VECTOR4D_PTR vpos, MATRIX4X4_PTR m)
{
// this function builds up a general local to world
// transformation matrix that is really nothing more than a translation
// of the origin by the amount specified in vpos

Mat_Init_4X4(m, 1,    0,    0,    0,
        0,    1,    0,    0,
        0,    0,    1,    0,
        vpos->x, vpos->y, vpos->z, 1 );

} // end Build_Model_To_World_MATRIX4X4
```

As you can see, the function takes the position to translate to and the matrix you want to store the computed transform matrix. Of course, you can use the matrix for general translation and not just the local-to-world transform. Anyway, to perform the local-to-world

Playrix Ex. 1023, page 679

10 0672318350 CH07 05•02•2003 2:42 PM Page 655

transformation with a matrix, we can make a call to `Build_Model_To_Model_MATRIX4X4()` to build up the matrix and then execute the transformation with `Transform_OBJECT4DV1()` like this:

```
MATRIX4X4 mt; // storage
VECTOR4D pos = {100,200,300, 1};

Build_Model_To_World_MATRIX4X4(&pos, &mt);

Transform_OBJECT4DV1(&obj, m_trans, TRANSFORM_LOCAL_TO_TRANS,1 );
```

So, that's all there is to performing the local-to-world transformation on `OBJECT4DV1`s.

NOTE

I know this is trivial stuff, but we're getting into good habits here—trust me….

Local-to-World Transformation for Render Lists

In most cases, you'll insert the polygons of an object into a render list after the object is already transformed from local to world coordinates. Thus, you'll probably never need to perform the local-to-world transform on an entire render list. But in the case when you don't want to use `OBJECT4DV1` and you load polygons directly into a `RENDERLIST4DV1` that happens to represent some large mesh (maybe a terrain, as Jonathan Blow, the world's expert on terrain rendering, would tell you), you'll need this step if the mesh isn't in world coordinates. Here's the function that manually performs the local to world transformation on a `RENDERLIST4DV1` without a matrix:

```
void Model_To_World_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
               POINT4D_PTR world_pos, int
               coord_select = TRANSFORM_LOCAL_TO_TRANS)
{
// NOTE: Not matrix based
// this function converts the local model coordinates of the
// sent render list into world coordinates, the results are stored
// in the transformed vertex list (tvlist) within the renderlist

// interate thru vertex list and transform all the model/local
// coords to world coords by translating the vertex list by
// the amount world_pos and storing the results in tvlist[]
// is this polygon valid?

if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
  {
  for (int poly = 0; poly < rend_list->num_polys; poly++)
    {
```


Playrix Ex. 1023, page 680

10 0672318350 CH07 05•02•2003 2:42 PM Page 656


```
    // acquire current polygon
    POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

    // transform this polygon if and only if it's not clipped,
    // not culled,
    // active, and visible,
    // note however the concept of "backface" is
    // irrelevant in a wire frame engine though
    if ((curr_poly==NULL) ¦¦
      !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
      (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
      (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
     continue; // move onto next poly

  // all good, let's transform
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // translate vertex
    VECTOR4D_Add(&curr_poly->vlist[vertex], world_pos,
          &curr_poly->tvlist[vertex]);
    } // end for vertex

  } // end for poly
} // end if local
else // TRANSFORM_TRANS_ONLY
{
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // acquire current polygon
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // transform this polygon if and only if it's
  // not clipped, not culled,
  // active, and visible, note however the concept of "backface" is
  // irrelevant in a wire frame engine though
 if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

  for (int vertex = 0; vertex < 3; vertex++)
    {
    // translate vertex
```

Playrix Ex. 1023, page 681

10 0672318350 CH07 05•02•2003 2:42 PM Page 657


```
    VECTOR4D_Add(&curr_poly->tvlist[vertex], world_pos,
          &curr_poly->tvlist[vertex]);
    } // end for vertex

  } // end for poly

} // end else

} // end Model_To_World_RENDERLIST4DV1
```

Thus, if you have a render list named `rend_list` and you want to perform the local-to-world transform and move the geometry from its local position to the position `world_pos`, here's what you would do:

```
Model_To_World_RENDERLIST4DV1(&rend_list, &world_pos, TRANSFORM_LOCAL_TO_TRANS);
```

That would be very efficient. But, if you wanted to use a matrix, you could do so by creating the transform like this:

```
MATRIX4X4 mt; // storage
VECTOR4D pos = {100,200,300, 1};

Build_Model_To_World_MATRIX4X4(&pos, &mt);
```

And then use the generic `RENDERLIST4DV1` function to transform the geometry like this:

```
Transform_RENDERLIST4DV1(&rend_list, &mt, TRANSFORM_LOCAL_TO_TRANS);
```


NOTE

But this method would be waste of cycles due to the overhead of a matrix multiply per each vertex—ouch!

The Eulerian Camera Model

This isn't really a step of the 3D pipeline, but this is a good time to talk about the camera models we're going to use, because the next steps in the pipeline rely on them. If you recall, the camera that we've been using for our first system is called *Eulerian*, meaning that it's defined by a position and the rotation (or Euler angles) that make up its orientation. This is shown in Figure 7.7. Additionally, we're going to support the *UVN* camera, which needs only the addition of a look at target, and the **u**,**v**,**n** vectors themselves—more on that later, though.

Playrix Ex. 1023, page 682

10 0672318350 CH07 05•02•2003 2:42 PM Page 658

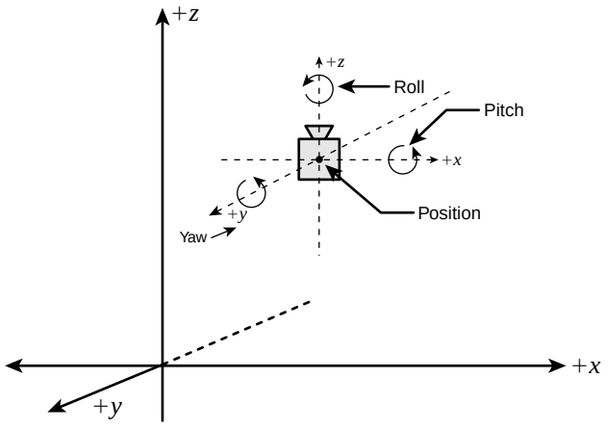


FIGURE 7.7 Euler camera model.

Whichever camera we use, we still need the viewing distance, a field of view, the near and far clipping planes, and a view plane of some width and height that represents the final image we want to display on the screen at some point, along with a viewport that has width and height and represents the actual raster window. All these aspects play into the camera model and create a viewing frustrum that encloses the camera's overall field of view and defines its properties. Objects within this field of view will be transformed and viewed.

Okay, so with all that in mind, I sat down and made up a structure to represent a simple camera. Really, I probably could have gotten away with the angles, position, near and far clipping planes, and a couple other things, but I wanted to start you off early thinking about all the information that a camera needs, even if we don't use it yet. Alas, there's some extra stuff that isn't really needed yet in this chapter—especially in relation to clipping. Anyway, here's the camera structure:

```
// camera version 1
typedef struct CAM4DV1_TYP
{
int state;   // state of camera
int attr;    // camera attributes

POINT4D pos;  // world position of camera used by both camera models

VECTOR4D dir;  // euler angles or look at direction of camera for UVN
        // models

VECTOR4D u;   // extra vectors to track the camera orientation
VECTOR4D v;   // for more complex UVN camera model
```

Playrix Ex. 1023, page 683

10 0672318350 CH07 05•02•2003 2:42 PM Page 659


```
VECTOR4D n;
POINT4D target; // target or "look at" position for UVN model


float view_dist_h; // horizontal and vertical viewing distances
float view_dist_v;

float fov;     // field of view for both horizontal and vertical axes

// 3d clipping planes
// if view volume is NOT 90 degree then general 3d clipping
// must be employed
float near_clip_z;   // near z=constant clipping plane
float far_clip_z;   // far z=constant clipping plane

PLANE3D rt_clip_plane; // the right clipping plane
PLANE3D lt_clip_plane; // the left clipping plane
PLANE3D tp_clip_plane; // the top clipping plane
PLANE3D bt_clip_plane; // the bottom clipping plane

float viewplane_width; // width and height of view plane to project onto
float viewplane_height; // usually 2x2 for normalized projection or
            // the exact same size as the viewport or
            // screen window

// remember screen and viewport are synonomous
float viewport_width;   // size of screen/viewport
float viewport_height;
float viewport_center_x; // center of view port (final image destination)
float viewport_center_y;

// aspect ratio
float aspect_ratio;   // screen aspect width/height

// these matrices are not necessarily needed based on the method of
// transformation, for example, a manual perspective or screen transform
// and or a concatenated perspective/screen, however, having these
// matrices give us more flexibility

MATRIX4X4 mcam; // storage for the world to camera transform matrix
MATRIX4X4 mper; // storage for the camera to perspective transform matrix
MATRIX4X4 mscr; // storage for the perspective to screen transform matrix

} CAM4DV1, *CAM4DV1_PTR;
```


Playrix Ex. 1023, page 684

10 0672318350 CH07 05•02•2003 2:42 PM Page 660

Although I commented the structure, let's review every single element so that there's no confision, I mean confusion!

`state`—Tracks the state of the camera. Nothing weird about that.

`attr`—Defines the particular attributes of the camera. Currently, there are only two type attributes: `CAM_MODEL_EULER` and `CAM_MODEL_UVN`.

`pos`—Defines the world position of the camera.

`dir`—Defines the axis angles for simple Eulerian camera model or, when using the UVN model, the x-component is the "elevation" for the spherical system and the y-component is the "heading."

`u,v,n`—Define vectors to track the camera orientation for the more complex UVN camera model we discuss later in the chapter.

`target`—Defines the target position that the UVN camera model looks at.

`view_dist_h`, `view_dist_v`—Define the horizontal and vertical viewing distances needed in the perspective transformation step.

`fov`—Defines the field of view for both horizontal and vertical axes.

`near_clip_z`, `far_clip_z`—Define the near and far constant z clipping planes.

`rt_clip_plane`, `lt_clip_plane`, `tp_clip_plane`, `bt_clip_plane`—Define the right, left, top, and bottom 3D clipping planes. For a field of view at 90 degrees, these planes are trivial at 45 degrees from the origin, but for arbitrary fields of view, these planes are no longer x = z, y = z, and so on, and must be generalized during 3D clipping.

`viewplane_width`, `viewplane_height`—Define the size of the viewing plane or viewplane that the perspective projection projects onto. This is 2x2 for normalized projection, or the exact same size as the view port or screen window if you want to merge the perspective transform and screen transform together (faster).

`viewport_width`, `viewport_height`—Define the size in pixel space of the final raster surface that you're rendering on.

`viewport_center_x`, `viewport_center_y`—Define the center of the viewport.

`aspect_ratio`—Simply the ratio of `viewport_width/viewport_height`.

`mcam`—Storage for the camera transformation only.

`mper`—Storage for the camera-to-perspective transformation.

`mscr`—Storage for the perspective-to-screen transformation.

Wow, a lot of stuff, huh? But remember, we aren't going to use or need all of it right away. The point of the structure is to hold all the camera information, but there's a lot of derived information in there that we don't want to have to define or compute ourselves.

Playrix Ex. 1023, page 685

10 0672318350 CH07 05•02•2003 2:42 PM Page 661

Thus, we need a function that can take a set of simple inputs and build a camera. First, let's look at the prototype of such a function:

```
void Init_CAM4DV1(CAM4DV1_PTR cam,    // the camera object
         int cam_attr,     // camera attributes
         POINT4D_PTR cam_pos,  // initial camera position
         VECTOR4D_PTR cam_dir, // initial camera angles
         POINT4D_PTR cam_target, // initial target for UVN model
         float near_clip_z,   // near and far clipping planes
         float far_clip_z,
         float fov,       // field of view in degrees
         float viewport_width, // size of final screen viewport
         float viewport_height);
```

With this function, we can send the type of camera, position, orientation, distance of the clipping planes, the field of view, and viewport dimensions, and the function sets up the whole camera for us! Now, the only problem is that many of these parameters are related, and hence constrained by each other.

Therefore, I wrote the function so that, in all cases, it creates a viewplane such that the width is {-1 to 1}, and the height will be {-1/`aspect_ratio` to +1/`aspect_ratio`}. This has to be agreed upon, because during the perspective to screen projection or mapping, we need to know what we're mapping. Forcing this convention makes it easier.

In most cases, we're going to use a 90-degree field of view, which in turn results in a focal distance of 1.0, but I wanted to add the extra functionality to the camera setup function just so you could experiment with FOVs that are different from 90 degrees. Lastly, I decided that the camera function should NOT compute the camera matrix during initialization. The camera matrix should be created at a later time via calls to either the Euler or UVN camera helper function. So, let's take a look at the UVN camera now.

The UVN Camera Model

The Eulerian camera is okay for many setups, but it's fairly limited and doesn't work exactly the way you think it does when you supply the view angles. A more natural camera model is the UVN camera. It's much easier to work with, so what we're going to do now is come up with the derivation of the camera model and the camera transformation matrix. Figure 7.8 shows the standard UVN system.

Basically, the only difference between a Euler camera and a UVN camera is how we define the orientation of the camera; instead of using angles, we use vectors. These vectors describe the orientation or the basis of the camera's coordinate system:

UVN = { **X**:<ux, uy, uz>, **Y**:<vx, vy, vz>, **Z**:<nx, ny, nz>}

For example, a standard XYZ left-handed system would have the following basis, using the preceding notation:

XYZ = { **X**:<1, 0, 0>, **Y**:<0, 1, 0>, **Z**:<0, 0, 1>}

Playrix Ex. 1023, page 686

10 0672318350 CH07 05•02•2003 2:42 PM Page 662

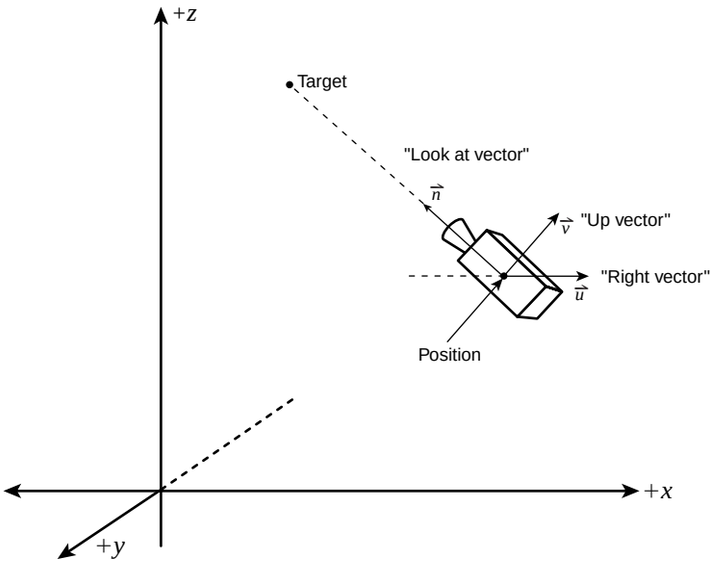


FIGURE 7.8 The UVN camera model.

The UVN system has the same properties as the XYZ system; it's mutually perpendicular, or *orthonormal*, and all the vectors are linearly independent and have unit length.

Additionally, the UVN system has what's called a *target* or *look at* point, which is nothing more than where the camera is looking. And, of course, the camera has a position in space that's called the *view reference point* (or VRP, in most cases).

Now let's talk about the UVN vectors and what they mean, but let's go a little out of order.

n—This vector is analogous to the z-axis of the UVN system and points toward the target position from the camera position. In other words, **n** = <target—VRP>. Of course, **n** must be a unit vector, so we would normalize this vector at some point.

v—This is called the "up" vector and begins life identical to the y-axis vector <0,1,0>. However, this value of **v** is only temporary and needed only to compute the "right" vector or **u**, and then a new **v** is computed.

u—This is called the "right" vector, and is computed from the cross product of **v** and **n** as **u** = (**v** x **n**). Here's where the temporary value of **v** comes in handy. Once **u** is computed, a recalculation of **v** is performed such that **v** = (**n** x **u**).

NOTE

Of course, all vectors must be normalized at some point.

Playrix Ex. 1023, page 687

10 0672318350 CH07 05•02•2003 2:42 PM Page 663

Assuming that we've now computed the **u**, **v**, and **n** vectors of the UVN system, all that's left is to understand what they do!

Well, it's pretty simple, actually. UVN is basically the new orientation of the XYZ system that we want to transform everything into; or, in other words, it's the basis of the system. Thus, we need to somehow create a transformation matrix that will perform the mapping. That is, assuming that a UVN camera is out in space at some position looking at a target, what matrix will perform this transform? First, let's forget about the position of the camera. I think we both know that's a simple translation, so the hard part is the rotation part….

Computing the UVN Matrix

It turns out that all we need to transform any point **p**(x,y,z) in the standard XYZ system into a point in the UVN system is to take the dot product of **p**(x, y, z) with **u**, **v**, and **n**, respectively. In other words, we need to multiply x by the amount that the u-axis is co-linear with the x, y by the amount the v-axis is colinear with the y-axis, and finally multiply z by the amount that the n-axis is colinear with the z-axis of the standard XYZ system. This will transform the point **p**(x,y,z) into the UVN space. Here's the matrix in 4x4 form:

$$\mathbf{M}_{uvn} = \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pretty slick, huh? So, the column vectors are just **u**,**v**, and **n**, respectively. Assuming that we take a point **p**(x,y,z,1) and transform it with this matrix to get **p'**, we get

$$\mathbf{p}' = \mathbf{p} * \mathbf{M}_{uvn}$$

$$= [x\ y\ z\ 1] * \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \mathbf{p}'((x*ux + y*uy + z*uz),\ (x*vx + y*vy + z*vz),\ (x*nx + y*ny + z*nz), 1)$$

Playrix Ex. 1023, page 688

10 0672318350 CH07 05•02•2003 2:42 PM Page 664

which is indeed the dot product terms we were looking for:

$\mathbf{p}' = (\mathbf{p} \cdot \mathbf{u}, \mathbf{p} \cdot \mathbf{v}, \mathbf{p} \cdot \mathbf{n})$

The last part of the puzzle is the positioning of the camera, but that's easy. We know that any camera transformation looks something like a translation step followed by a rotational transformation.

$\mathbf{T}_{cam}^{-1} * \mathbf{R}_{cam}^{-1}$

where $\mathbf{R}_{cam}^{-1}$ is a composition of one or more rotational transformations.

> **NOTE**
>
> The inverses are just to remind you that in the original derivation, we had the matrix that positioned the camera. To undo that, we needed the inverse of that matrix to bring the universe to the camera, and similarly for the rotation matrices.

We already have the rotational transform that we called $\mathbf{M}_{uvn}$, so all we need is the translation matrix. Assuming that the camera is located at position `cam_pos=(cam_x, cam_y, cam_z)` as usual, this is nothing more than

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

Thus, we end up with the final transformation for the UVN camera of

$\mathbf{T}_{uvn} = \mathbf{T}_{cam}^{-1} * \mathbf{M}_{uvn}$

=

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix} * \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

=

$$\begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ -(cam\_pos \cdot u) & -(cam\_pos \cdot v) & -(cam\_pos \cdot n) & 1 \end{bmatrix}$$

Playrix Ex. 1023, page 689

10 0672318350 CH07 05•02•2003 2:42 PM Page 665

The terms in the last row are the translation factors; `tx`, `ty`, and `tz`, which are nothing more than the negatives of the dot product of the camera position with each of the basis vectors **u**, **v**, and **n**, respectively.

That's all there is to the UVN camera system, so let's briefly review and then move on to how to actually compute and describe the UVN system.

The UVN system models the camera as usual via a camera position, but differs in that the camera has a target. Additionally, instead of representing the camera as a series of concatenated rotations parallel to each of the axes in some order, XYZ, YZX, and so on, the camera in a UVN system is made of a "look at" vector, **n**, an "up" vector, **v**, and finally a "right" vector, **u**. These correlate to the standard XYZ axes with the following mapping:

U -> X

V -> Y

N -> Z

Here are the exact steps to find **u**, **v**, and **n**:

Step 1: n = <target position—view reference point>

Step 2: Let **v** = <0,1,0>

Step 3: u = (**v** x **n**)

Step 4: v = (**n** x **u**)

Step 5 (Optional): Normalize **u**,**v**,**n** by dividing each vector by its length (if you haven't done so in previous steps)

Practical Implementations of the UVN System

As you can see, the UVN system is pretty cool and very natural to the way you might expect to use a camera in the real world. That is, you position the camera and then target a position (or subject) to view, as shown in Figure 7.8. The only problem is that many times you don't exactly know the coordinates of the position you're looking at, but know the general direction. Or, in other words, a more Eulerian model is appropriate, but as we've previously discovered, the Euler camera model doesn't always do what we want and suffers from gimbal lock and other issues. Therefore, if you know where you want to look, you can directly compute **u**, **v**, and **n** and be done with it. However, what about a more natural camera system based on a spherical system, where you give the position of the camera and then the "heading" and "elevation" (as shown in Figure 7.9), and the math computes **u**, **v**, and **n** for you? This isn't as hard as it seems. Take a look at Figure 7.9; it illustrates the setup we need.

Playrix Ex. 1023, page 690

10 0672318350 CH07 05•02•2003 2:42 PM Page 666

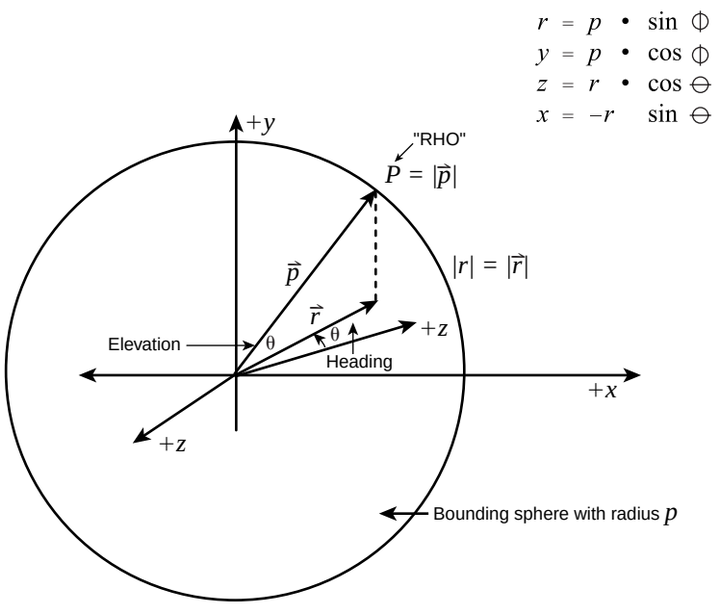


FIGURE 7.9 The spherical setup for the UVN system.

First, we assume that the camera is sitting at the origin (enclosed in a unit sphere) to make the derivation easier. Obviously, to move the camera anywhere, we just add in the translation step. Referring to the figure, the angular heading θ (theta) is the counter-clockwise angle around the positive y-axis with the convention that the 0 degree angle occurs when the heading is aligned with the positive z-axis. Next, the elevation ϕ (phi) is the angle measured from the ground plane (the x-z plane) to the direction vector, as shown in the figure. Now, based on our discussion of spherical coordinates, we usually refer to a point in spherical coordinates as **p**(ρ,ϕ,θ). However, in the derivation in Chapter 4, I assumed a right-handed system along with other conventions about the angles and so forth, so we want to be careful not to use the exact same formulas from Chapter 4. They need to be slightly modified as follows:

Given the spherical camera position **p**(ρ,ϕ,θ):

P = The target vector

ρ = The length of the target vector, which in this case is 1.0 because we're in a unit sphere

θ = The heading angle

ϕ = The elevation angle

r = The length of the projection of the target vector on the x-z plane

Playrix Ex. 1023, page 691

10 0672318350 CH07 05•02•2003 2:42 PM Page 667

FORMULA 7.1 Computation of UVN Target Vector from Spherical Coordinates

then

$r = \rho * \sin(\phi)$

Therefore

$x = -r * \sin(\theta)$

$y = \rho * \cos(\phi)$

$z = r * \cos(\theta)$

If you look at the result from the function `SPHERICAL3D_To_POINT3D()`, the x,y,z it generates are based on a right-handed system. To map them to our system, we need the following transformation:

$x = -y$

$y = z$

$z = x$

Based on the computed value of **p**(x,y,z), this is exactly the look-at target, and thus we can completely define a UVN camera with just the heading, elevation, and position of the camera because we can compute the rest.

Well, that's it for the UVN system. With the Euler camera and the UVN system, you should be able to create any kind of camera you like. Shortly, we'll see the code that sets up cameras and the different calls to create the camera matrices for both systems.

Initializing the Camera

As I mentioned, the camera function isn't going to set up the actual camera transformation matrix (that will be handled by the helper functions), but is going to set up everything else having to do with the viewplane, clipping planes, and so forth. Here's the function:

```
void Init_CAM4DV1(CAM4DV1_PTR cam,    // the camera object
         int cam_attr,     // camera attributes
         POINT4D_PTR cam_pos,  // initial camera position
         VECTOR4D_PTR cam_dir, // initial camera angles
         POINT4D_PTR cam_target, // initial target for UVN model
         float near_clip_z,   // near and far clipping planes
         float far_clip_z,
         float fov,       // field of view in degrees
         float viewport_width, // size of final screen viewport
         float viewport_height)
{
```

Playrix Ex. 1023, page 692

10 0672318350 CH07 05•02•2003 2:42 PM Page 668


```
// this function initializes the camera object cam, the function
// doesn't do a lot of error checking or sanity checking since
// I want to allow you to create projections as you wish, also
// I tried to minimize the number of parameters the functions needs

// first set up parms that are no brainers
cam->attr = cam_attr;       // camera attributes

VECTOR4D_COPY(&cam->pos, cam_pos); // positions
VECTOR4D_COPY(&cam->dir, cam_dir); // direction vector or angles for
                  // euler camera

// for UVN camera
VECTOR4D_INITXYZ(&cam->u, 1,0,0); // set to +x
VECTOR4D_INITXYZ(&cam->v, 0,1,0); // set to +y
VECTOR4D_INITXYZ(&cam->n, 0,0,1); // set to +z

if (cam_target!=NULL)
  VECTOR4D_COPY(&cam->target, cam_target); // UVN target
else
  VECTOR4D_ZERO(&cam->target);

cam->near_clip_z = near_clip_z;   // near z=constant clipping plane
cam->far_clip_z = far_clip_z;   // far z=constant clipping plane

cam->viewport_width = viewport_width;  // dimensions of viewport
cam->viewport_height = viewport_height;

cam->viewport_center_x = (viewport_width-1)/2; // center of viewport
cam->viewport_center_y = (viewport_height-1)/2;

cam->aspect_ratio = (float)viewport_width/(float)viewport_height;

// set all camera matrices to identity matrix
MAT_IDENTITY_4X4(&cam->mcam);
MAT_IDENTITY_4X4(&cam->mper);
MAT_IDENTITY_4X4(&cam->mscr);

// set independent vars
cam->fov       = fov;

// set the viewplane dimensions up, they will be 2 x (2/ar)
cam->viewplane_width = 2.0;
cam->viewplane_height = 2.0/cam->aspect_ratio;
```

Playrix Ex. 1023, page 693

10 0672318350 CH07 05•02•2003 2:42 PM Page 669


```
// now we know fov and we know the viewplane dimensions plug into formula and
// solve for view distance parameters
float tan_fov_div2 = tan(DEG_TO_RAD(fov/2));

cam->view_dist = (0.5)*(cam->viewplane_width)*tan_fov_div2;

// test for 90 fov first since it's easy :)
if (fov == 90.0)
  {
   // set up the clipping planes -- easy for 90 degrees!
   POINT3D pt_origin; // point on the plane
   VECTOR3D_INITXYZ(&pt_origin,0,0,0);

   VECTOR3D vn; // normal to plane

   // right clipping plane
   VECTOR3D_INITXYZ(&vn,1,0,-1); // x=z plane
   PLANE3D_Init(&cam->rt_clip_plane, &pt_origin, &vn, 1);

   // left clipping plane
   VECTOR3D_INITXYZ(&vn,-1,0,-1); // -x=z plane
   PLANE3D_Init(&cam->lt_clip_plane, &pt_origin, &vn, 1);

   // top clipping plane
   VECTOR3D_INITXYZ(&vn,0,1,-1); // y=z plane
   PLANE3D_Init(&cam->tp_clip_plane, &pt_origin, &vn, 1);

   // bottom clipping plane
   VECTOR3D_INITXYZ(&vn,0,-1,-1); // -y=z plane
   PLANE3D_Init(&cam->bt_clip_plane, &pt_origin, &vn, 1);
  } // end if d=1
else
  {
   // now compute clipping planes yuck!
   POINT3D pt_origin; // point on the plane
   VECTOR3D_INITXYZ(&pt_origin,0,0,0);

   VECTOR3D vn; // normal to plane

   // since we don't have a 90 fov, computing the normals
   // are a bit tricky, there are a number of geometric constructions
   // that solve the problem, but I'm going to solve for the
   // vectors that represent the 2D projections of the frustrum planes
   // on the x-z and y-z planes and then find perpendiculars to them
```


Playrix Ex. 1023, page 694

10 0672318350 CH07 05•02•2003 2:42 PM Page 670


```
    // right clipping plane, check the math on graph paper
    VECTOR3D_INITXYZ(&vn,cam->view_dist,0,-cam->viewplane_width/2.0);
    PLANE3D_Init(&cam->rt_clip_plane, &pt_origin, &vn, 1);

    // left clipping plane, we can simply reflect the right normal about
    // the z axis since the planes are symmetric about the z axis
    // thus invert x only
    VECTOR3D_INITXYZ(&vn,-cam->view_dist,0,-cam->viewplane_width/2.0);
    PLANE3D_Init(&cam->lt_clip_plane, &pt_origin, &vn, 1);

    // top clipping plane, same construction
    VECTOR3D_INITXYZ(&vn,0,cam->view_dist,-cam->viewplane_width/2.0);
    PLANE3D_Init(&cam->tp_clip_plane, &pt_origin, &vn, 1);

    // bottom clipping plane, same inversion
    VECTOR3D_INITXYZ(&vn,0,-cam->view_dist,-cam->viewplane_width/2.0);
    PLANE3D_Init(&cam->bt_clip_plane, &pt_origin, &vn, 1);
   } // end else

} // end Init_CAM4DV1
```

Please take some time to review the function. It's not that complicated at all, but it's a good example of the problems with cameras and projections and working around constraints.

Anyway, the function first determines what you're specifying and then, based on the field of view and the final viewport size, it computes everything based on that using the x-axis as the primary axis that must be equal to the selected field of view. Note that the function doesn't really care much about the camera model that's selected at this point. It just remembers it and sets up a couple fields, but for the most part, the camera helper functions build the actual camera matrix `mcam`. Now, let's take a look at some quick examples.

Example 1: Parameterization for Euler camera with normalized coordinates camera setup with FOV = 90, viewing distance 1.0, and screen dimension 400 × 400. The function computes the viewplane size, which in this case would be 2x2. Figure 7.10 shows what this camera model would look like.

```
// initialize the camera
Init_CAM4DV1(&cam,   // the camera object
       CAM_MODEL_EULER, // the camera model
       &cam_pos, // initial camera position
       &cam_dir, // initial camera angles
       NULL,   // not used in euler, send NULL
       50.0,   // near and far clipping planes
       500.0,
```

Playrix Ex. 1023, page 695

10 0672318350 CH07 05•02•2003 2:42 PM Page 671


```
90.0,   // field of view in degrees
400,  // size of final screen viewport
400);
```

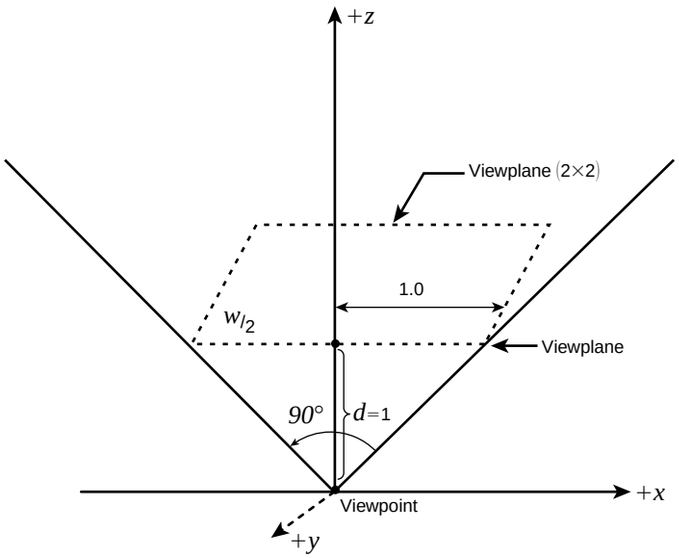


FIGURE 7.10 Camera setup with normalized coordinates, FOV of 90 with viewing distance 1.0.

Example 2: Parameterization for UVN camera with a FOV = 90 with the viewport 640x480, for a simplified screen projection. The function computes the viewing distance for us. Figure 7.11 shows what this camera model would look like.

```
// initialize the camera
Init_CAM4DV1(&cam,   // the camera object
       CAM_MODEL_UVN, // the camera type
       &cam_pos, // initial camera position
       &cam_dir, // initial camera angles
       &cam_target, // initial target
       50.0,   // near and far clipping planes
       500.0,
       90.0,   // field of view in degrees
       640,  // size of final screen viewport
       480);
```

Playrix Ex. 1023, page 696

10 0672318350 CH07 05•02•2003 2:42 PM Page 672

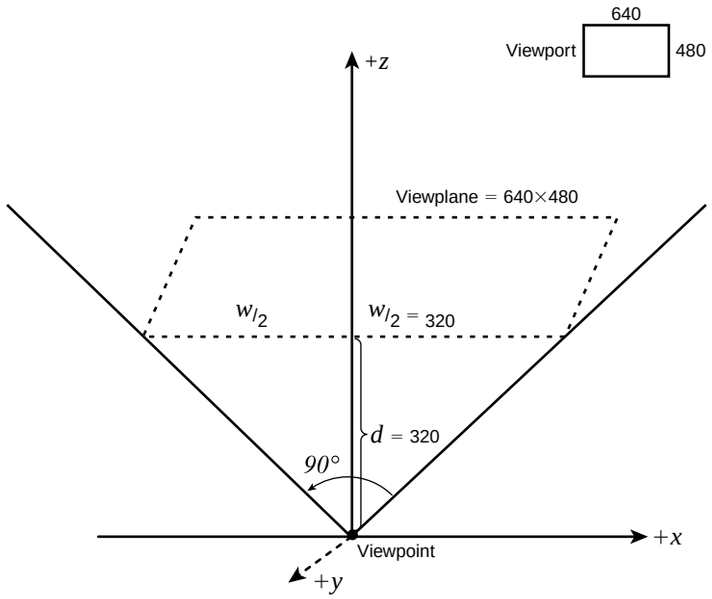


FIGURE 7.11 Camera setup for FOV of 90 with viewport dimensions equal to viewplane.

In both cases, the viewplane, focal distance, and aspect ratio are computed internally. In general, the setups from Examples 1 and 2 will suffice for all of our work. We always want FOV = 90 (or maybe 60), and we want either a nice normalized projection onto a viewplane of 2x2, or 2x2/`aspect_ratio` for non-square viewports.

Alright, we're almost done with the camera stuff. The last thing we need is a way to position the camera and compute the matrices in the camera object. So, here's a function that you can call with a position and the Euler angles that will take a camera object and compute all the matrices and other elements for you:

```
void Build_CAM4DV1_Matrix_Euler(CAM4DV1_PTR cam, int cam_rot_seq)
{
// this creates a camera matrix based on Euler angles
// and stores it in the sent camera object
// if you recall from chapter 6 to create the camera matrix
// we need to create a transformation matrix that looks like:

// Mcam = mt(-1) * my(-1) * mx(-1) * mz(-1)
// that is the inverse of the camera translation matrix mutilplied
// by the inverses of yxz, in that order, however, the order of
// the rotation matrices is really up to you, so we aren't going
// to force any order, thus its programmable based on the value
```

Playrix Ex. 1023, page 697

10 0672318350 CH07 05•02•2003 2:42 PM Page 673


```
// of cam_rot_seq which can be any value CAM_ROT_SEQ_XYZ where
// XYZ can be in any order, YXZ, ZXY, etc.

MATRIX4X4 mt_inv, // inverse camera translation matrix
     mx_inv, // inverse camera x axis rotation matrix
     my_inv, // inverse camera y axis rotation matrix
     mz_inv, // inverse camera z axis rotation matrix
     mrot,  // concatenated inverse rotation matrices
     mtmp;  // temporary working matrix


// step 1: create the inverse translation matrix for the camera
// position
Mat_Init_4X4(&mt_inv, 1,  0,   0,   0,
           0,  1,   0,   0,
           0,  0,   1,   0,
           -cam->pos.x, -cam->pos.y, -cam->pos.z, 1);

// step 2: create the inverse rotation sequence for the camera
// rember either the transpose of the normal rotation matrix or
// plugging negative values into each of the rotations will result
// in an inverse matrix

// first compute all 3 rotation matrices

// extract out euler angles
float theta_x = cam->dir.x;
float theta_y = cam->dir.y;
float theta_z = cam->dir.z;

// compute the sine and cosine of the angle x
float cos_theta = Fast_Cos(theta_x); // no change since cos(-x) = cos(x)
float sin_theta = -Fast_Sin(theta_x); // sin(-x) = -sin(x)

// set the matrix up
Mat_Init_4X4(&mx_inv, 1,  0,     0,     0,
           0,  cos_theta, sin_theta, 0,
           0,  -sin_theta, cos_theta, 0,
           0,  0,     0,     1);

// compute the sine and cosine of the angle y
cos_theta = Fast_Cos(theta_y); // no change since cos(-x) = cos(x)
sin_theta = -Fast_Sin(theta_y); // sin(-x) = -sin(x)
```

Playrix Ex. 1023, page 698

10 0672318350 CH07 05•02•2003 2:42 PM Page 674


```
// set the matrix up
Mat_Init_4X4(&my_inv,cos_theta, 0, -sin_theta, 0,
           0,     1, 0,     0,
           sin_theta, 0, cos_theta, 0,
           0,     0, 0,     1);

// compute the sine and cosine of the angle z
cos_theta = Fast_Cos(theta_z); // no change since cos(-x) = cos(x)
sin_theta = -Fast_Sin(theta_z); // sin(-x) = -sin(x)

// set the matrix up
Mat_Init_4X4(&mz_inv, cos_theta, sin_theta, 0, 0,
           -sin_theta, cos_theta, 0, 0,
           0,     0,     1, 0,
           0,     0,     0, 1);

// now compute inverse camera rotation sequence
switch(cam_rot_seq)
   {
   case CAM_ROT_SEQ_XYZ:
   {
   Mat_Mul_4X4(&mx_inv, &my_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
   } break;

   case CAM_ROT_SEQ_YXZ:
   {
   Mat_Mul_4X4(&my_inv, &mx_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
   } break;

   case CAM_ROT_SEQ_XZY:
   {
   Mat_Mul_4X4(&mx_inv, &mz_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &my_inv, &mrot);
   } break;

   case CAM_ROT_SEQ_YZX:
   {
   Mat_Mul_4X4(&my_inv, &mz_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
   } break;
```

Playrix Ex. 1023, page 699

10 0672318350 CH07 05•02•2003 2:42 PM Page 675


```
   case CAM_ROT_SEQ_ZYX:
   {
   Mat_Mul_4X4(&mz_inv, &my_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
   } break;

   case CAM_ROT_SEQ_ZXY:
   {
   Mat_Mul_4X4(&mz_inv, &mx_inv, &mtmp);
   Mat_Mul_4X4(&mtmp, &my_inv, &mrot);

   } break;

   default: break;
   } // end switch

// now mrot holds the concatenated product of inverse rotation matrices
// multiply the inverse translation matrix against it and store in the
// camera objects' camera transform matrix we are done!
Mat_Mul_4X4(&mt_inv, &mrot, &cam->mcam);

} // end Build_CAM4DV1_Matrix_Euler
```

The function takes two parameters: the camera object itself and a flag that controls the rotation sequence for the Euler angles. The angles themselves and the position of the camera are internal to the camera, so you just set them in the camera object before making this call. Anyway, the various settings for the rotation control sequence are

```
// defines for camera rotation sequences
#define CAM_ROT_SEQ_XYZ 0
#define CAM_ROT_SEQ_YXZ 1
#define CAM_ROT_SEQ_XZY 2
#define CAM_ROT_SEQ_YZX 3
#define CAM_ROT_SEQ_ZYX 4
#define CAM_ROT_SEQ_ZXY 5
```

Assume that you've already created a camera called `cam` and initialized it with a call to `Init_CAM4DV1()`. To tell the camera to build up its matrices for a ZYX rotation sequence, here's the call:

```
// set position
cam.pos.x = 100;
cam.pos.y = 200;
cam.pos.z = 300;
```

Playrix Ex. 1023, page 700

10 0672318350 CH07 05•02•2003 2:42 PM Page 676


```
// set viewing angles (degrees)
cam.dir.x = -45;
cam.dir.y = 0;
cam.dir.z = 0;

// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

After the call, the matrix `cam.mcam` would be valid and you could use it as the world-to-camera transformation. Figure 7.12 shows the position and orientation of the virtual camera with these settings. However, the matrices `cam.mper` and `cam.mscr` are still undefined.

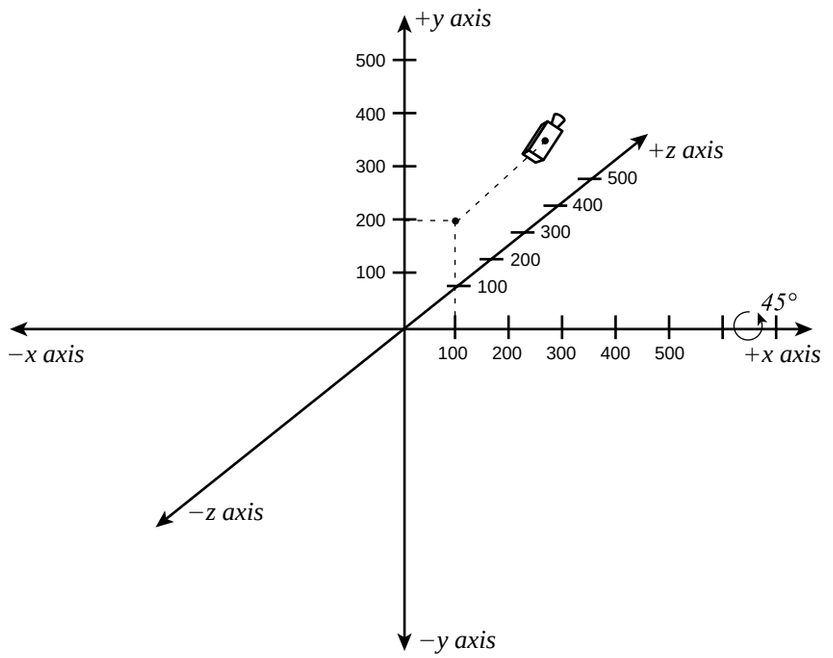


FIGURE 7.12 The results of the camera setup call.

Next, here's the camera function for the UVN model:

```
void Build_CAM4DV1_Matrix_UVN(CAM4DV1_PTR cam, int mode)
{
// this creates a camera matrix based on a look at vector n,
// look up vector v, and a look right (or left) u
// and stores it in the sent camera object, all values are
```

Playrix Ex. 1023, page 701

10 0672318350 CH07 05•02•2003 2:42 PM Page 677


```
// extracted out of the camera object itself
// mode selects how uvn is computed
// UVN_MODE_SIMPLE - low level simple model,
// use the target and view reference point
// UVN_MODE_SPHERICAL - spherical mode, the x,y components will be used as the
//   heading and elevation of the view vector respectively
//   along with the view reference point as the position
//   as usual

MATRIX4X4 mt_inv, // inverse camera translation matrix
     mt_uvn, // the final uvn matrix
     mtmp;  // temporary working matrix

// step 1: create the inverse translation matrix for the camera
// position
Mat_Init_4X4(&mt_inv, 1,  0,   0,   0,
           0,  1,   0,   0,
           0,  0,   1,   0,
           -cam->pos.x, -cam->pos.y, -cam->pos.z, 1);


// step 2: determine how the target point will be computed
if (mode == UVN_MODE_SPHERICAL)
  {
  // use spherical construction
  // target needs to be recomputed

  // extract elevation and heading
  float phi  = cam->dir.x; // elevation
  float theta = cam->dir.y; // heading

  // compute trig functions once
  float sin_phi = Fast_Sin(phi);
  float cos_phi = Fast_Cos(phi);

  float sin_theta = Fast_Sin(theta);
  float cos_theta = Fast_Cos(theta);

  // now compute the target point on a unit sphere x,y,z
  cam->target.x = -1*sin_phi*sin_theta;
  cam->target.y = 1*cos_phi;
  cam->target.z = 1*sin_phi*cos_theta;
  } // end else
```


Playrix Ex. 1023, page 702

10 0672318350 CH07 05•02•2003 2:42 PM Page 678


```
// at this point, we have the view reference point, the target and that's
// all we need to recompute u,v,n
// Step 1: n = <target position - view reference point>
VECTOR4D_Build(&cam->pos, &cam->target, &cam->n);

// Step 2: Let v = <0,1,0>
VECTOR4D_INITXYZ(&cam->v,0,1,0);

// Step 3: u = (v x n)
VECTOR4D_Cross(&cam->v,&cam->n,&cam->u);

// Step 4: v = (n x u)
VECTOR4D_Cross(&cam->n,&cam->u,&cam->v);

// Step 5: normalize all vectors
VECTOR4D_Normalize(&cam->u);
VECTOR4D_Normalize(&cam->v);
VECTOR4D_Normalize(&cam->n);


// build the UVN matrix by placing u,v,n as the columns of the matrix
Mat_Init_4X4(&mt_uvn, cam->u.x,  cam->v.x,   cam->n.x,   0,
           cam->u.y,  cam->v.y,   cam->n.y,   0,
           cam->u.z,  cam->v.z,   cam->n.z,   0,
           0,      0,      0,      1);

// now multiply the translation matrix and the uvn matrix and store in the
// final camera matrix mcam
Mat_Mul_4X4(&mt_inv, &mt_uvn, &cam->mcam);

} // end Build_CAM4DV1_Matrix_UVN
```

Alright, we can finally get back the 3D pipeline—that's what we call a *stall* <BG>.

The World-to-Camera Transform

Now that we have the camera modeled and set up, we can perform the world-to-camera transformation, but let's quickly review why the camera building functions work. Both the UVN and the Euler model do nothing more than take some representation of where the camera is and what it's orientation is and then build a matrix. This boils down to two steps: a translation step and a rotational step.

Figure 7.13 illustrates the steps in the world-to-camera transformation. The 3D world is viewed via a virtual camera positioned at some point with some orientation and with a

Playrix Ex. 1023, page 703

10 0672318350 CH07 05•02•2003 2:42 PM Page 679

given horizontal and vertical field of view. Through the lens of this virtual camera, we're interested in what the viewer would see if the camera and world were real. The trick is to create a transformation matrix that position transforms all the objects in the world so that we could artificially position the camera at the origin (0,0,0) looking straight down the +z axis. This way the math would be simplified for the final perspective projection later in the pipeline.

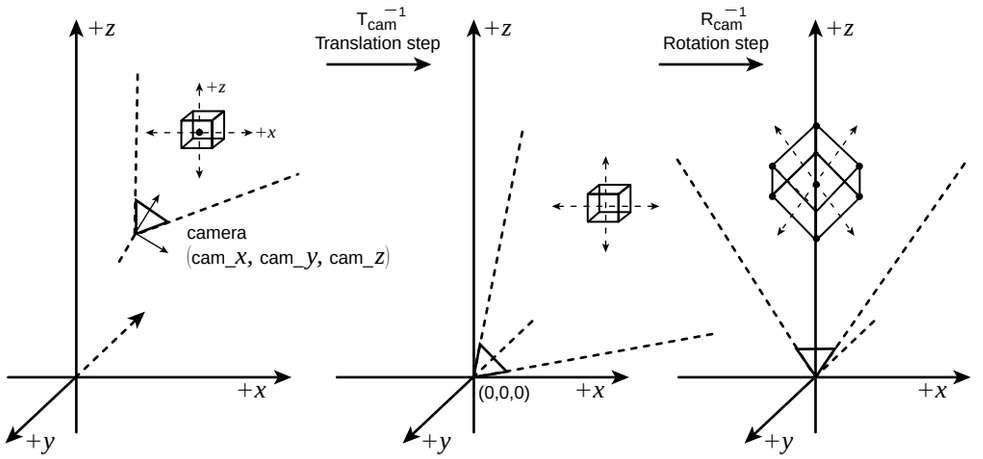


FIGURE 7.13 The world-to-camera transformation "reloaded."

We learned that this trick could be accomplished in the following way: If we assume that the camera is located in world space at `cam_pos` = *(*`cam_x, cam_y, cam_z`*)* defined by the transformation $\mathbf{T}_{cam}$, step one would be to translate all objects, vertices, polygons, or whatever in the world by the inverse of this matrix (value) or `-cam_pos` = *(*`-cam_x, -cam_y, -cam_z`*)*; that is, $\mathbf{T}_{cam}^{-1}$. The matrix for it looks like this:

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

Transforming each vertex by this matrix translates all the geometry relative to the camera being at the origin. Now, the next step in the camera transformation is the rotation step. Of course, this is a little trickier, and the model of the camera enters into the final 4x4 matrix that represents the camera orientation. With the Eulerian camera model we have been using, three angles to represent the orientation of the camera: `cam_ang` = *(*`ang_x, ang_y, ang_z`*)* as shown in Figure 7.14. Each angle represents the clockwise or counter-clockwise (right-handed system) rotation around each positive axis. We call the separate transformations $\mathbf{R}_{camy}$, $\mathbf{R}_{camx}$, $\mathbf{R}_{camz}$. However, if you recall, the order in which you apply

Playrix Ex. 1023, page 704

10 0672318350 CH07 05•02•2003 2:42 PM Page 680

these rotations is important; XYZ, YZX, and so forth. But we decided to use **YXZ** and **ZYX** for most cases.

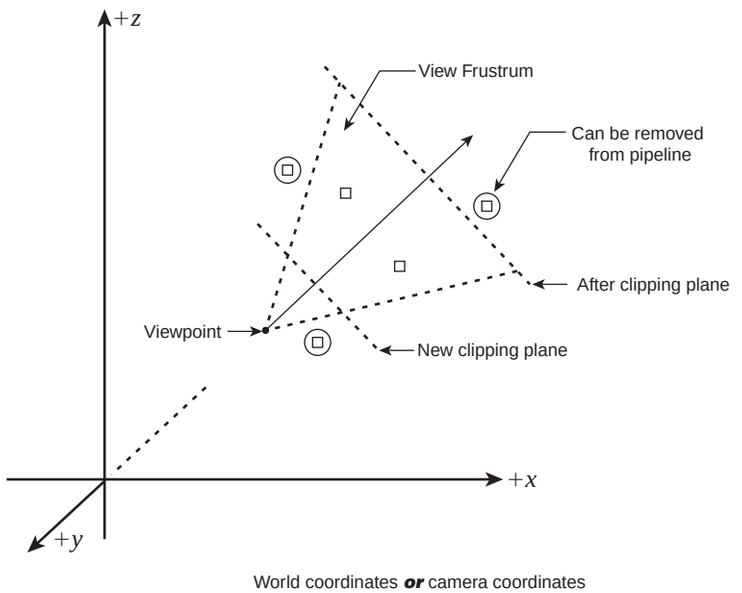


FIGURE 7.14 Relationship of camera rotation angles to coordinate systems.

So, to perform the rotation step of the camera transformation we need to apply the inverse rotations or the sequence:

$$\mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$$

or maybe

$$\mathbf{R}_{camz}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1}$$

Putting this all together, the camera transform is $\mathbf{T}_{wc}$:

$$\mathbf{T}_{wc} = \mathbf{T}_{cam}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$$

Or, if you want the ZYX rotation order:

$$\mathbf{T}_{wc} = \mathbf{T}_{cam}^{-1} * \mathbf{R}_{camz}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camz}^{-1}$$

Whichever you desire, the point is that you need a translation followed by three rotations. And, of course, these operations can be concatenated into a single matrix called $\mathbf{T}_{wc}$ as shown previously. All you do is multiply each vertex that you want to transform by the

Playrix Ex. 1023, page 705

10 0672318350 CH07 05•02•2003 2:42 PM Page 681

matrix, and that's it. That's what all the camera software does: It builds up this transform for you. All you do is transform each vertex by the matrix $\mathbf{T}_{wc}$ and you're done with the world-to-camera transformation! We already have a function to build the camera and initialize it, and one to compute the world-to-camera transformation matrix `Build_CAM4DV1_Matrix_Euler()`. And, of course, the `Build_CAM4DV1_Matrix_UVN()` function does the same thing, it just does it differently. That is, it uses the concept of the up, right, and target vectors rather than angles, but in the end, both functions arrive at a transformation that we need to apply to every vertex.

So, all we need is a function to apply the transform to an object and/or render list. Now, the interesting thing is that we already have a pair of generic transformation functions that take any matrix and can transform an object or render list: `void Transform_OBJECT4DV1()` and `void Transform_RENDERLIST4DV1()`. We can use either of those functions and walk away, but I like to write specific functions—sometimes for each step in the pipeline—instead of using the generalized ones so that I can take advantage of optimizations, so we'll see both.

World-to-Camera Transformation for Objects

As mentioned, we have the ability to apply any 4x4 transform matrix to an `OBJECT4DV1` with a call to `Transform_OBJECT4DV1()`, so let's look at that approach first. Assuming that we have an `object obj`, `camera cam`, and everything is initialized:

```
// generate camera matrix (use euler cam)
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// now apply the transform and make sure to use the
// world coordinates and transform them in place
Transform_OBJECT4DV1(&obj, &cam.mcam, TRANSFORM_TRANS_ONLY, 1 );
```

That would do the trick. The object would be in camera space all ready for a perspective projection and screen mapping. On the other hand, we can code the function a little more manually instead of using the `Transform_OBJECT4DV1()` function. Here's such a coding:

```
void World_To_Camera_OBJECT4DV1(CAM4DV1_PTR cam, OBJECT4DV1_PTR obj)
{
// NOTE: this is a matrix based function
// this function transforms the world coordinates of an object
// into camera coordinates, based on the sent camera matrix
// but it totally disregards the polygons themselves,
// it only works on the vertices in the vlist_trans[] list
// this is one way to do it, you might instead transform
// the global list of polygons in the render list since you
// are guaranteed that those polys represent geometry that
// has passed thru backfaces culling (if any)
```

Playrix Ex. 1023, page 706

10 0672318350 CH07 05•02•2003 2:42 PM Page 682


```
// transform each vertex in the object to camera coordinates
// assumes the object has already been transformed to world
// coordinates and the result is in vlist_trans[]
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // transform the vertex by the mcam matrix within the camera
  // it better be valid!
  POINT4D presult; // hold result of each transformation

  // transform point
  Mat_Mul_VECTOR4D_4X4(&obj->vlist_trans[vertex], &cam->mcam, &presult);

  // store result back
  VECTOR4D_COPY(&obj->vlist_trans[vertex], &presult);
  } // end for vertex

} // end World_To_Camera_OBJECT4DV1
```

This function is a little faster and a little tighter. It takes the object and the camera as parameters and that's it. Here's the call you would make given that you have the object and camera already set up:

```
World_To_Camera_OBJECT4DV1(&cam, &obj);
```

> **NOTE**
>
> In a real 3D software engine, I might even hard-code the matrix and vector operations, but the rule of thumb is that doing so everywhere in your entire engine may give you 5–10%. If you need it, fine; otherwise, it's not worth the clutter. Better to design clean algorithms rather than to optimize code at that level during phase I design.

World-to-Camera Transformation for Render Lists

Performing the world-to-camera transform with the rendering is the same as with objects. We can use the generic transform function or write a manual function to transform the render list (call it `rend_list`). Again, let's do both. Here's the method using what we already have:

```
// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// now apply the transform and make sure to use the
// world coordinates and transform them in place
Transform_RENDERLIST4DV1(&rend_list, &cam.mcam, TRANSFORM_TRANS_ONLY);
```

Playrix Ex. 1023, page 707

10 0672318350 CH07 05•02•2003 2:42 PM Page 683

Note that the `transform_basis` flag is not part of the parameter list to `Transform_RENDERLIST4DV1()`. This is because the polygon list represents many objects, and thus the orientation of these objects is irrelevant once the polygons have been extracted from each object and merged. Moving on, here's a manual function that performs the transformation. The only complexity is that each polygon of the list must be traversed, whereas in the `OBJECT4DV1` version of the manual function, only a single vertex list had to be transformed. Here's the function:

```
void World_To_Camera_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                CAM4DV1_PTR cam)
{
// NOTE: this is a matrix based function
// this function transforms each polygon in the global render list
// to camera coordinates based on the sent camera transform matrix
// you would use this function instead of the object based function
// if you decided earlier in the pipeline to turn each object into
// a list of polygons and then add them to the global render list
// the conversion of an object into polygons probably would have
// happened after object culling, local transforms, local to world
// and backface culling, so the minimum number of polygons from
// each object are in the list, note that the function assumes
// that at LEAST the local to world transform has been called
// and the polygon data is in the transformed list tvlist of
// the POLYF4DV1 object

// transform each polygon in the render list into camera coordinates
// assumes the render list has already been transformed to world
// coordinates and the result is in tvlist[] of each polygon object

for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// acquire current polygon
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// is this polygon valid?
// transform this polygon if and only if it's not clipped, not culled,
// active, and visible, note however the concept of "backface" is
// irrelevant in a wire frame engine though
if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly
```


Playrix Ex. 1023, page 708

10 0672318350 CH07 05•02•2003 2:42 PM Page 684


```
// all good, let's transform
for (int vertex = 0; vertex < 3; vertex++)
  {
  // transform the vertex by the mcam matrix within the camera
  // it better be valid!
  POINT4D presult; // hold result of each transformation

  // transform point
  Mat_Mul_VECTOR4D_4X4(&curr_poly->tvlist[vertex], &cam->mcam, &presult);

  // store result back
  VECTOR4D_COPY(&curr_poly->tvlist[vertex], &presult);
  } // end for vertex

} // end for poly

} // end World_To_Camera_RENDERLIST4DV1
```

And here's the call that would transform an entire render list `rend_list`, assuming that `cam` is defined and initialized as before:

```
World_To_Camera_RENDERLIST4DV1(&cam, &rend_list);
```

Simplataco! Yes, that's a word <BG>. At this point, we're about half way down the pipeline.

Now all we need is the perspective transformation projection and the screen projection, and that's it. However, there are a couple caveats…. First, we've said nothing about clipping, and that's a big issue. However, we did cover in the previous chapter that clipping can be performed in 3D object space, or in 2D raster image space; alas, we're going to have to do something about it.

However, we definitely can't wait all the way until rasterization because we surely can't project polygon vertices with negative or, worse yet, 0.0 z-values (divide by 0.0 error)! So, keep that in mind. We have to deal with that somewhat, but right now, let's talk about removing whole objects from the viewing pipeline.

Object Culling

Object culling is just that: removing objects so that they don't have to be processed through the entire 3D pipeline, as shown in Figure 7.15. However, object culling can be performed in world space *or* camera space. Nonetheless, I prefer to do it in camera space. "But," you might ask, "if you wait all the way until camera space to perform the culling of an object, didn't you already waste all those cycles transforming the object into camera space to see if it's invisible?" And the answer is, not necessarily.

Playrix Ex. 1023, page 709

10 0672318350 CH07 05•02•2003 2:42 PM Page 685

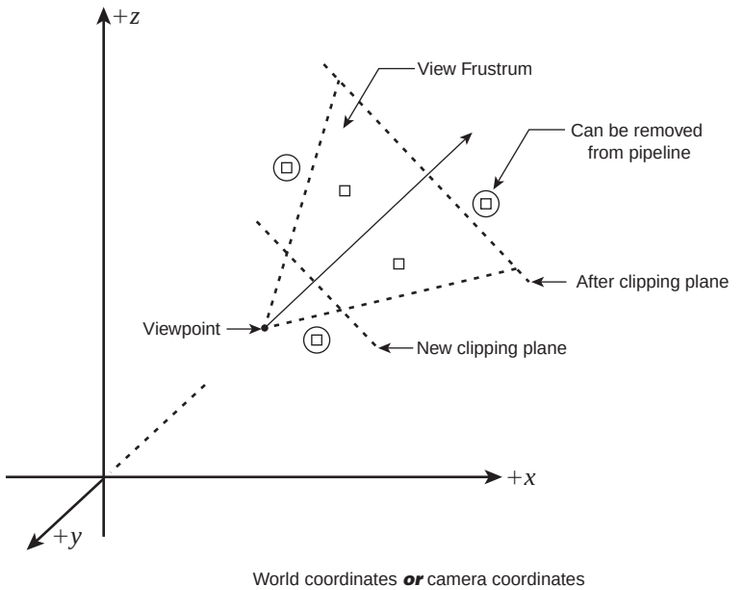


FIGURE 7.15 Removing objects from the 3D pipeline.

There's a trick. If you use the center of the object and its maximum radius to create a bounding sphere, you can perform the object culling test by transforming a single point from world-to-camera coordinates that represents the center of the object being tested. Then you basically test the bounding sphere for inclusion/exclusion in the viewing frustrum. This is detailed in the previous chapter, but if you recall, the math ended up being nothing more than testing a few points against the right, left, top, bottom, near, and far clipping planes. In any case, this is one of the *most important* steps in a 3D engine pipeline because in most cases, only a few objects will be in the scene at a time; the rest will be out of range or behind you, so there's no reason to process them. Therefore, we definitely want to add this functionality to our engine pipeline.

To accomplish the object culling step, we need a function that takes the object's current world position along with it's average/maximum radius (you might use either). The function then culls the object based on the current camera transformation. This only works for objects because the sense of "object" is lost once we convert an object into its constituent polygons and insert them into the master rendering list. Anyway, here's the object removal function:

```
int Cull_OBJECT4DV1(OBJECT4DV1_PTR obj, // object to cull
          CAM4DV1_PTR cam,   // camera to cull relative to
          int cull_flags)   // clipping planes to consider
{
```

Playrix Ex. 1023, page 710

10 0672318350 CH07 05•02•2003 2:42 PM Page 686


```
// NOTE: is matrix based
// this function culls an entire object from the viewing
// frustrum by using the sent camera information and object
// the cull_flags determine what axes culling should take place
// x, y, z or all which is controlled by ORing the flags
// together
// if the object is culled its state is modified thats all
// this function assumes that both the camera and the object
// are valid!

// step 1: transform the center of the object's bounding
// sphere into camera space

POINT4D sphere_pos; //hold result of transforming cntr of bounding sphere

// transform point
Mat_Mul_VECTOR4D_4X4(&obj->world_pos, &cam->mcam, &sphere_pos);

// step 2: based on culling flags remove the object
if (cull_flags & CULL_OBJECT_Z_PLANE)
{
// cull only based on z clipping planes

// test far plane
if ( ((sphere_pos.z - obj->max_radius) > cam->far_clip_z) ¦¦
   ((sphere_pos.z + obj->max_radius) < cam->near_clip_z) )
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if

} // end if

if (cull_flags & CULL_OBJECT_X_PLANE)
{
// cull only based on x clipping planes
// we could use plane equations, but simple similar triangles
// is easier since this is really a 2D problem
// if the view volume is 90 degrees the the problem is trivial
// buts lets assume its not

// test the right and left clipping planes
// against the leftmost and rightmost
```

Playrix Ex. 1023, page 711

10 0672318350 CH07 05•02•2003 2:42 PM Page 687


```
// points of the bounding sphere
float z_test = (0.5)*cam->viewplane_width*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.x-obj->max_radius) > z_test) ¦¦ // right side
   ((sphere_pos.x+obj->max_radius) < -z_test) ) // left side
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if
} // end if

if (cull_flags & CULL_OBJECT_Y_PLANE)
{
// cull only based on y clipping planes
// we could use plane equations, but simple similar triangles
// is easier since this is really a 2D problem
// if the view volume is 90 degrees the the problem is trivial
// buts lets assume its not

// test the the top and bottom clipping planes
// against the bottommost and topmost
// points of the bounding sphere
float z_test = (0.5)*cam->viewplane_height*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.y-obj->max_radius) > z_test) ¦¦ // top side
   ((sphere_pos.y+obj->max_radius) < -z_test) ) // bottom side
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if

} // end if

// return failure to cull
return(0);

} // end Cull_OBJECT4DV1
```

The function is a straightforward mathematical implementation of our findings on object culling in the last chapter, so there's no need to reiterate that. However, there are some interesting additions to the function. Basically, the function takes the object to test for culling, the current camera (you must make a call to initialize the `mcam` matrix before using the function), and a flags parameter `cull_flags` that defines the suite of tests to perform. Let's look at that....

Playrix Ex. 1023, page 712

10 0672318350 CH07 05•02•2003 2:42 PM Page 688

The parameter `cull_flags` controls the axes that culling will be performed on. For example, if you want culling only on the z-axis (near and far clip planes), an object might still be invisible on the x and y axes, but at least you're guaranteed not to project polygons from an object that's behind you. In general, you should perform culling on all six faces of the 3D viewing frustrum because the tests are so cheap. Anyway, the flags that control the culling are as follows:

```
// general culling flags
#define CULL_OBJECT_X_PLANE   0x0001 // cull on the x clipping planes
#define CULL_OBJECT_Y_PLANE   0x0002 // cull on the y clipping planes
#define CULL_OBJECT_Z_PLANE   0x0004 // cull on the z clipping planes
#define CULL_OBJECT_XYZ_PLANES  (CULL_OBJECT_X_PLANE ¦ CULL_OBJECT_Y_PLANE /
                    ¦ CULL_OBJECT_Z_PLANE)
```

You just logically `OR` them together for what you want. For example, here's how you would cull an object to the entire frustrum:

```
if (Cull_OBJECT4DV1(&obj, &cam,
           CULL_OBJECT_X_PLANE ¦
           CULL_OBJECT_Y_PLANE ¦
           CULL_OBJECT_Z_PLANE))
{ /* object culled */ }
```

After culling, the function will alter the object's state flags and set a bit to indicate that the object was culled. To facilitate this, I've defined some constants for `OBJECT4DV1` states:

```
// states for objects
#define OBJECT4DV1_STATE_ACTIVE      0x0001
#define OBJECT4DV1_STATE_VISIBLE     0x0002
#define OBJECT4DV1_STATE_CULLED      0x0004
```

We'll undoubtedly add to this, but for now, by setting and then testing the `OBJECT4DV1_STATE_CULLED` in rendering and transforming functions, we can disregard objects that have been culled. Additionally, an object may or may not be culled each frame. So, in each game frame, you need to reset your object's flags so that all the transient flags (such as the culling flag, for instance) are cleared. Here's a simple function that does that:

```
void Reset_OBJECT4DV1(OBJECT4DV1_PTR obj)
{
// this function resets the sent object and readies it for
// transformations, basically just resets the culled, clipped and
// backface flags, but here's where you would add stuff
// to ready any object for the pipeline
// the object is valid, let's rip it apart polygon by polygon
```

Playrix Ex. 1023, page 713

10 0672318350 CH07 05•02•2003 2:42 PM Page 689


```
// reset object's culled flag
RESET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);

// now the clipped and backface flags for the polygons
for (int poly = 0; poly < obj->num_polys; poly++)
  {
  // acquire polygon
  POLY4DV1_PTR curr_poly = &obj->plist[poly];

  // first is this polygon even visible?
  if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE))
    continue; // move onto next poly

  // reset clipped and backface flags
  RESET_BIT(curr_poly->state, POLY4DV1_STATE_CLIPPED);
  RESET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

  } // end for poly

} // end Reset_OBJECT4DV1
```

Making a call to the function resets the object's culling flag as well as resetting the back-face flag for each polygon that makes up the object's polygon mesh—let's talk about that next.

Back-face Removal

Back-face removal is shown in Figure 7.16. It's the process of removing polygons that are facing *away* from the viewpoint. We covered the mathematics of this in the last chapter, so I'm not going to go into detail. But in a nutshell, the test is performed on every single polygon of the object or render list. A direction vector is computed from the polygon to the viewpoint and the angle between this and the outward normal of the polygon is computed. If the angle is greater than 90 degrees or $\mathbf{n} \cdot \mathbf{l} <= 0$, the polygon is a back-face and can't possibly be visible.

Of course, the concept of back-face only makes sense for polygons that have one side. For polygons that are visible from both sides (two-sided), this test doesn't help us. Anyway, the test is usually performed in world space (because all we need is the viewpoint), not camera space, because we can usually kill a lot of polygons with back-face removal before pushing them through the world-to-camera transform, so it's a good way to remove another 50% of your geometry!

Playrix Ex. 1023, page 714

10 0672318350 CH07 05•02•2003 2:42 PM Page 690

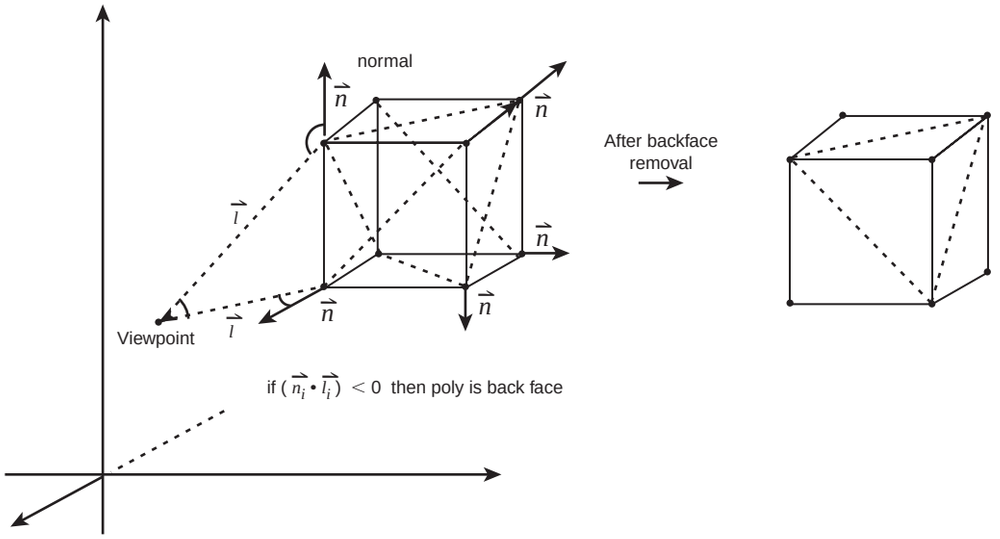


FIGURE 7.16 Backface removal in action.

Removing Back-faces in Objects

Here's the function that removes back-faces for objects contained in the `OBJECT4DV1` structure:

```
void Remove_Backfaces_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function removes the backfaces from an object's
// polygon mesh, the function does this based on the vertex
// data in vlist_trans along with the camera position (only)
// note that only the backface state is set in each polygon

// test if the object is culled
if (obj->state & OBJECT4DV1_STATE_CULLED)
  return;

// process each poly in mesh
for (int poly=0; poly < obj->num_polys; poly++)
  {
  // acquire polygon
  POLY4DV1_PTR curr_poly = &obj->plist[poly];
```

Playrix Ex. 1023, page 715

10 0672318350 CH07 05•02•2003 2:42 PM Page 691


```
// is this polygon valid?
// test this polygon if and only if it's not clipped, not culled,
// active, and visible and not 2 sided.
// Note we test for backface in the event that
// a previous call might have already determined this, so why work
// harder!
if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
   (curr_poly->attr & POLY4DV1_ATTR_2SIDED)  ¦¦
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
  continue; // move onto next poly

// extract vertex indices into master list, remember the polygons are
// NOT self contained, but based on the vertex list stored in the object
// itself
int vindex_0 = curr_poly->vert[0];
int vindex_1 = curr_poly->vert[1];
int vindex_2 = curr_poly->vert[2];

// we will use the transformed polygon vertex list since the backface removal
// only makes sense at the world coord stage further of the pipeline

// we need to compute the normal of this polygon face, and recall
// that the vertices are in cw order, u = p0->p1, v=p0->p2, n=uxv
VECTOR4D u, v, n;

// build u, v
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_1 ], &u);
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
        &obj->vlist_trans[ vindex_2 ], &v);

// compute cross product
VECTOR4D_Cross(&u, &v, &n);

// now create eye vector to viewpoint
VECTOR4D view;
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ], &cam->pos, &view);

// and finally, compute the dot product
float dp = VECTOR4D_Dot(&n, &view);
```


Playrix Ex. 1023, page 716

10 0672318350 CH07 05•02•2003 2:42 PM Page 692


```
  // if the sign is > 0 then visible, 0 = scathing, < 0 invisible
  if (dp <= 0.0 )
    SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

  } // end for poly

} // end Remove_Backfaces_OBJECT4DV1
```

Aren't you glad we wrote all those math and vector functions? I am! They made this function really easy to write. Anyway, take a few minutes to look at the function and see if it makes sense. Basically, it takes the object, traverses the polygon list, and tests whether all polygons that are active, visible, and one-sided are back-faces from the current viewpoint as defined in the camera object. Note that you don't need the `mcam` matrix defined for this function because all it needs is the camera's position. Here's how you would use the function:

```
Remove_Backfaces_OBJECT4DV1(&obj,&cam);
```

Hard, huh? And whamo! the polygon list in the object would have its flags altered for all back-faces. Specifically, each polygon that is a back-face gets its back-face bit set like this:

```
SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);
```

TIP

Make sure that you perform back-face removal *after* object removal, but *before* the world-to-camera transformation.

Removing Back-faces in Render Lists

Now, if you decided to keep your objects in one piece before back-face removal and/or you just have a polygon list and no objects in your universe, you'll want to perform back-face removal on the render list. Of course, if you have objects and you performed this step previously and then inserted them into the rendering list, there's no point. Anyway, the function that tests for back-faces in a `RENDERLIST4DV1` object is almost identical to the `OBJECT4DV1` version, it's just a little different data structure access. Here it is:

```
void Remove_Backfaces_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list, CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function removes the backfaces from polygon list
// the function does this based on the polygon list data
// tvlist along with the camera position (only)
// note that only the backface state is set in each polygon
```

Playrix Ex. 1023, page 717

10 0672318350 CH07 05•02•2003 2:42 PM Page 693


```
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // acquire current polygon
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // is this polygon valid?
  // test this polygon if and only if it's not clipped, not culled,
  // active, and visible and not 2 sided.
  // Note we test for backface in the event that
  // a previous call might have already determined this, so why work
  // harder!
  if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
    (curr_poly->attr & POLY4DV1_ATTR_2SIDED)  ¦¦
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

    // we need to compute the normal of this polygon face, and recall
    // that the vertices are in cw order, u = p0->p1, v=p0->p2, n=uxv
    VECTOR4D u, v, n;

    // build u, v
    VECTOR4D_Build(&curr_poly->tvlist[0], &curr_poly->tvlist[1], &u);
    VECTOR4D_Build(&curr_poly->tvlist[0], &curr_poly->tvlist[2], &v);

    // compute cross product
    VECTOR4D_Cross(&u, &v, &n);

    // now create eye vector to viewpoint
    VECTOR4D view;
    VECTOR4D_Build(&curr_poly->tvlist[0], &cam->pos, &view);

    // and finally, compute the dot product
    float dp = VECTOR4D_Dot(&n, &view);

    // if the sign is > 0 then visible, 0 = scathing, < 0 invisible
    if (dp <= 0.0 )
      SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

     } // end for poly

} // end Remove_Backfaces_RENDERLIST4DV1
```


Playrix Ex. 1023, page 718

10 0672318350 CH07 05•02•2003 2:42 PM Page 694

Similar to the OBJECT4DV1 version, you want to remove back-faces before the world-to-camera transform because the transform is a waste if the polygons are back-faces and invisible! Finally, here's a sample call:

```
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);
```

After the call, the POLY4DV1_STATE_BACKFACE bit would be set in all polygons that were back-faces.

The Camera-to-Perspective Transform

The next stage in the 3D pipeline is the camera-to-perspective transformation. Figure 7.17 depicts the mathematics of the projection once again for your review. Basically, given a point $\mathbf{p} = (x_{world}, y_{world}, z_{world}, 1)$, the basic perspective transform is

```
x_per = viewing_distance*x_world/z_world
y_per = viewing_distance*aspect_ratio*y_world/z_world
```

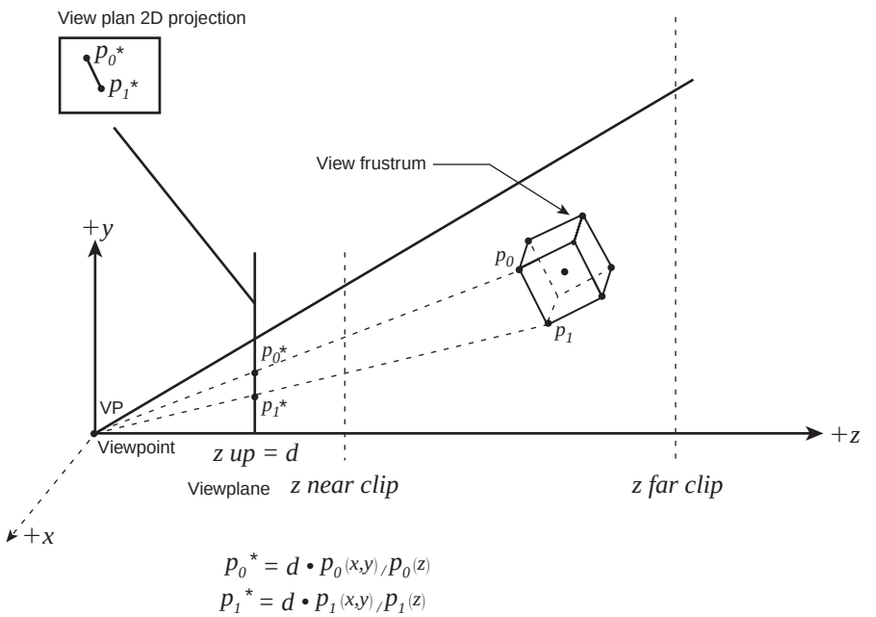


FIGURE 7.17 Review of perspective projection.

You recall that if we set the viewing distance (or d, for short) to 1.0, we ended up with normalized viewplane coordinates for the x and y axes in the range of

Playrix Ex. 1023, page 719

10 0672318350 CH07 05•02•2003 2:42 PM Page 695

x-axis—{x: –1 to 1}

y-axis—{y: -1/ar to 1/ar}

where ar is the aspect ratio width/height of the raster screen.

Additionally, a viewing distance of 1.0 forms a 90-degree field of view for both the x and y axes if the viewplane is square. Moving on, if we want to control the field of view or the viewing distance more, and project onto a viewplane of any size, we realized that viewing distance and field of view were dependent and we ended up with these formulas for the perspective transform:

$d = (0.5)*viewplane\_width*\tan(\theta_h/2)$

$xper = (d*x_{world})/z_{world}$
$yper = (d*y_{world}*aspect\_ratio)/z_{world}$

Alright, enough review. The point of all this is to implement the transform with code and we have two different approaches: (1) perform the math manually (faster), or (2) perform the transform using matrices. But the problem with method 2 and using matrix transforms, as you recall, is that there's no way to perform a divide by z at the end via matrices. Thus, we must use 4D homogeneous coordinates and then convert the results from the matrix transform back to 3D non-homogeneous coordinates if we want to use a 4x4 matrix transform. The fact that the matrix multiplication is wasteful in addition to the post-homogeneous conversion makes using a matrix for the perspective transform a bad thing from a performance standpoint in software. But once again, many hardware implementations like only matrices. With all that in mind, let's take a look at both methods for the OBJECT4DV1 type along with the RENDERLIST4DV1 type.

The Perspective Transform for Objects

The camera object has all the information that we need to perform the perspective transform, including the viewing distance, the viewplane width and height, and the field of view (which is used to compute the viewing distance), so there's not much to the transform except performing the couple of lines of transformation math to each vertex. Here's the function that performs this on OBJECT4DV1:

```
void Camera_To_Perspective_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms the camera coordinates of an object
// into perspective coordinates, based on the
// sent camera object, but it totally disregards the polygons themselves,
// it only works on the vertices in the vlist_trans[] list
// this is one way to do it, you might instead transform
// the global list of polygons in the render list since you
// are guaranteed that those polys represent geometry that
```

Playrix Ex. 1023, page 720

10 0672318350 CH07 05•02•2003 2:42 PM Page 696


```
// has passed thru backfaces culling (if any)
// finally this function is really for experimental reasons only
// you would probably never let an object stay intact this far down
// the pipeline, since it's probably that there's only a single polygon
// that is visible! But this function has to transform the whole mesh!

// transform each vertex in the object to perspective coordinates
// assumes the object has already been transformed to camera
// coordinates and the result is in vlist_trans[]
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  float z = obj->vlist_trans[vertex].z;

  // transform the vertex by the view parameters in the camera
  obj->vlist_trans[vertex].x = cam->view_dist_h *
                obj->vlist_trans[vertex].x/z;
  obj->vlist_trans[vertex].y = cam->view_dist_v *
                obj->vlist_trans[vertex].y *
                cam->aspect_ratio/z;
  // z = z, so no change

  // note that we are NOT dividing by the homogenous w coordinate since
  // we are not using a matrix op for this version of the function

  } // end for vertex

} // end Camera_To_Perspective_OBJECT4DV1
```

As you can see, the function is very short. Basically, it multiplies each vertex by the viewing distance (on both the x and y axes, which technically will be the same, but the aspect ratio takes the non-square viewplanes into consideration) and then divides by z. Here's a sample call:

```
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);
```

NOTE

The function does not test for z values that are negative or 0.0. This is up to you to handle in a previous culling or clipping step. But for now, assume that the vertices are all projectable.

Now, what if you want to use a matrix to perform the projection? No problem. We can use our general `Transform_OBJECT4DV1()` function with a perspective matrix, which we can build with the following function:

Playrix Ex. 1023, page 721

10 0672318350 CH07 05•02•2003 2:42 PM Page 697


```
void Build_Camera_To_Perspective_MATRIX4X4(CAM4DV1_PTR cam,
                      MATRIX4X4_PTR m)
{
// this function builds up a camera to perspective transformation
// matrix, in most cases the camera would have a 2x2 normalized
// view plane with a 90 degree FOV, since the point of the having
// this matrix must be to also have a perspective to screen (viewport)
// matrix that scales the normalized coordinates, also the matrix
// assumes that you are working in 4D homogenous coordinates and at
// some point there will be a 4D->3D conversion, it might be immediately
// after this transform is applied to vertices, or after the perspective
// to screen transform

Mat_Init_4X4(m,
cam->view_dist_h, 0,                0, 0,
0,        cam->view_dist_v*cam->aspect_ratio, 0, 0,
0,        0,                1, 1,
0,        0,                0,0);

} // end Build_Camera_To_Perspective_MATRIX4X4
```

Then we just apply the transform to the `OBJECT4DV1` like this (assuming that there's a camera object, of course):

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);
Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY,1);
```

Of course, there's one detail, right? After the transformation, the w factor in each vertex [x y z w] is no longer 1.0; thus, the coordinates are now completely homogeneous and we must convert them back to non-homogeneous before we use them. This is accomplished with a simple division by the w component of each vertex. Here's a function that performs the function on `OBJECT4DV1`:

```
void Convert_From_Homogeneous4D_OBJECT4DV1(OBJECT4DV1_PTR obj)
{
// this function converts all vertices in the transformed
// vertex list from 4D homogeneous coordinates to normal 3D coordinates
// by dividing each x,y,z component by w

for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
```

Playrix Ex. 1023, page 722

10 0672318350 CH07 05•02•2003 2:42 PM Page 698


```
  // convert to non-homogenous coords
  VECTOR4D_DIV_BY_W(&obj->vlist_trans[vertex]);
  } // end for vertex

} // end Convert_From_Homogeneous4D_OBJECT4DV1
```

So, as the last step to get real 3D coordinates, you just make the call:

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

The Perspective Transform for Render Lists

The perspective transform for the `RENDERLIST4DV1` type works identically to the one for `OBJECT4DV1`. The functions more or less work the same, but must access the data structures a little differently—that's about it. Here's the primary non-matrix camera to perspective transform function:

```
void Camera_To_Perspective_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                        CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms each polygon in the global render list
// into perspective coordinates, based on the
// sent camera object,
// you would use this function instead of the object based function
// if you decided earlier in the pipeline to turn each object into
// a list of polygons and then add them to the global render list

// transform each polygon in the render list into camera coordinates
// assumes the render list has already been transformed to world
// coordinates and the result is in tvlist[] of each polygon object

for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// acquire current polygon
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// is this polygon valid?
// transform this polygon if and only if it's not clipped, not culled,
// active, and visible, note however the concept of "backface" is
// irrelevant in a wire frame engine though
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly
```

Playrix Ex. 1023, page 723

10 0672318350 CH07 05•02•2003 2:42 PM Page 699


```
// all good, let's transform
for (int vertex = 0; vertex < 3; vertex++)
  {
  float z = curr_poly->tvlist[vertex].z;

  // transform the vertex by the view parameters in the camera
  curr_poly->tvlist[vertex].x = cam->view_dist_h *
                curr_poly->tvlist[vertex].x/z;
  curr_poly->tvlist[vertex].y = cam->view_dist_v *
                curr_poly->tvlist[vertex].y *
                cam->aspect_ratio/z;
  // z = z, so no change

  // note that we are NOT dividing by the homogenous w coordinate since
  // we are not using a matrix op for this version of the function

  } // end for vertex

} // end for poly

} // end Camera_To_Perspective_RENDERLIST4DV1
```

It's used in the same as the `OBJECT4DV1` version:

```
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);
```

Next, to implement the matrix version, we once again build the matrix with a call to `Build_Camera_To_Perspective_MATRIX4X4()` and then make a call to `Transform_RENDERLIST4DV1()` like this:

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_RENDERLIST4DV1(&rend_list, &mper, TRANSFORM_TRANS_ONLY);
```

You guessed it: We need to dehomogenize the coordinates because w is no longer 1.0 (actually w = z). Hence, here's the call to convert the homogeneous 4D back to 3D:

```
void Convert_From_Homogeneous4D_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list)
{
// this function convertes all valid polygons vertices in the transformed
// vertex list from 4D homogeneous coordinates to normal 3D coordinates
// by dividing each x,y,z component by w
```

Playrix Ex. 1023, page 724

10 0672318350 CH07 05•02•2003 2:42 PM Page 700


```
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// acquire current polygon
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// is this polygon valid?
// transform this polygon if and only if it's not clipped, not culled,
// active, and visible, note however the concept of "backface" is
// irrelevant in a wire frame engine though
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

// all good, let's transform
for (int vertex = 0; vertex < 3; vertex++)
  {
  // convert to non-homogenous coords
  VECTOR4D_DIV_BY_W(&curr_poly->tvlist[vertex]);
  } // end for vertex

} // end for poly

} // end Convert_From_Homogeneous4D_RENDERLIST4DV1
```

The final call sequence would of course look like this:

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_RENDERLIST4DV1(&rend_list, &mper, TRANSFORM_TRANS_ONLY);

Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);
```

At this point, we have a virtual image sitting on the viewplane, so to speak. Now we need to project it onto the viewscreen or viewport, and that's the last step of the process (other than the actual rasterization, of course).

The Perspective-to-Screen Viewport Transform

The perspective-to-screen transform is more or less the last stage of the 3D pipeline. It basically takes the viewplane coordinates and scales them to screen coordinates, as shown in Figure 7.18. The transform must take into consideration that most raster screens have the origin (0,0) at the top-left corner, and the y-axis is inverted from the standard 2D Cartesian coordinate system. Of course, if during the perspective transform itself the view-

Playrix Ex. 1023, page 725

10 0672318350 CH07 05•02•2003 2:42 PM Page 701

plane was made the same size as the viewport, a scaling operation will not be needed. But a translation and inversion of the y-axis still will be needed in most cases, because during projection we assume that the center of the viewplane is the origin at (0,0) and positive x is to the right, whereas positive y is upward. Raster screens, of course, have their origin (0,0) at the upper-left corner and the y-axis is inverted. Therefore, no matter what you do, you're going to need a viewport transform of some kind.

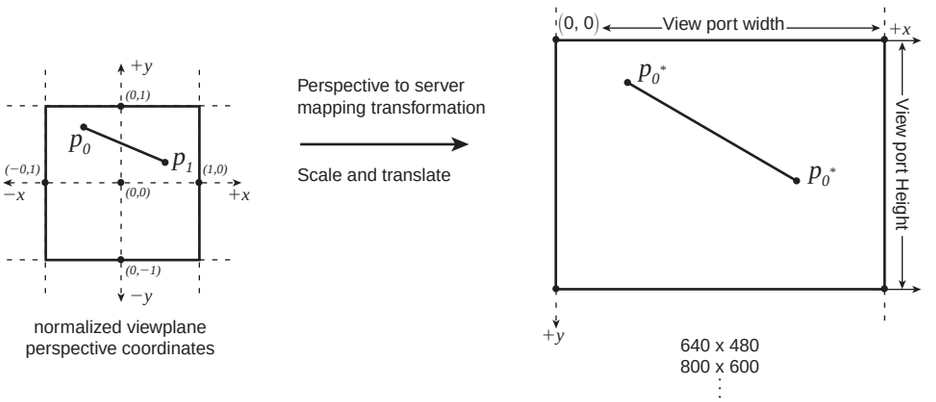


FIGURE 7.18 Transforming from perspective coordinates to screen coordinates.

With that in mind, let's take a look at a number of viewport transforms that I've written for both objects and render lists. But keep in mind that these transforms assume that the perspective transform was a normalizing one; that is, onto a viewplane of 2x2 for square projections, or 2/ar x 2/ar for non-square projections, so a scaling operation must be done. In the next section, we'll see the entire perspective/screen transform performed in a single step.

7

Viewport Transformation for Objects

First, we'll look at a manual function that performs the perspective to screen/viewport transformation. The function assumes that the perspective coordinates are in normalized form. So, the function must scale the coordinates to the size of the screen/viewport, and translate and invert the y-axis (of course, this is all in Chapter 6). Here's the function:

```
void Perspective_To_Screen_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms the perspective coordinates of an object
// into screen coordinates, based on the sent viewport info
// but it totally disregards the polygons themselves,
// it only works on the vertices in the vlist_trans[] list
// this is one way to do it, you might instead transform
```

Playrix Ex. 1023, page 726

10 0672318350 CH07 05•02•2003 2:42 PM Page 702


```
// the global list of polygons in the render list since you
// are guaranteed that those polys represent geometry that
// has passed thru backfaces culling (if any)
// finally this function is really for experimental reasons only
// you would probably never let an object stay intact this far down
// the pipeline, since it's probably that there's only a single polygon
// that is visible! But this function has to transform the whole mesh!
// this function would be called after a perspective
// projection was performed on the object

// transform each vertex in the object to screen coordinates
// assumes the object has already been transformed to perspective
// coordinates and the result is in vlist_trans[]

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // assumes the vertex is in perspective normalized coords from -1 to 1
  // on each axis,scale them to viewport and invert y axis and project
  // to screen

  // transform the vertex by the view parameters in the camera
  obj->vlist_trans[vertex].x = alpha + alpha*obj->vlist_trans[vertex].x;
  obj->vlist_trans[vertex].y = beta - beta *obj->vlist_trans[vertex].y;

  } // end for vertex

} // end Perspective_To_Screen_OBJECT4DV1
```

The function really just maps the coordinates {x:–1 to 1} and {y:-1 to 1} (normalized viewplane coordinates) to the raster screen. However, there's one detail that we should mention: the 4D homogeneous conversion is unnecessary, because during the camera-to-perspective transform, there was a division by z. Thus, after this transform the (x,y) component of each vertex is ready to go and represents the real position on the raster screen to plot.

Now, it's possible that during the perspective transform we did not divide by z; that is, maybe we used a matrix formulation and left the coordinates in 4D homogeneous form (where w is not equal to 1.0). We can still use this function, but at some point we must divide each vertex by its w component to convert from homogeneous to non-homogeneous. However, in most cases, you don't leave the coordinate data in 4D form all the way

Playrix Ex. 1023, page 727

10 0672318350 CH07 05•02•2003 2:42 PM Page 703

down to the viewport transform, but there's no rule against doing it. Anyway, here's how you would use the function in the context of the perspective object transform:

```
// first camera to perspective transform
// after transform coordinates are already non-homogeneous
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);

// now perform final viewport transform
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);
```

Almost too easy!

Now let's talk about the perspective-to-screen transform using matrices. Here's the clincher: The matrix will be different depending on whether the perspective coordinates are still in 4D homogeneous form or if they've already been divided by w or z (depending on the transform was performed) and are the real 3D perspective coordinates (technically 2D; the x,y is all we need, though). So, let's take a look at both 4D and 3D versions.

Let's assume that we performed the perspective transform with this code:

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY,1);
```

The results would be in 4D form. Thus, to find the final coordinates, we need to divide each vertex by its w value. However, we can hold off on that and still perform the perspective-to-screen viewport transform in 4D coordinates with this matrix-building function:

```
void Build_Perspective_To_Screen_4D_MATRIX4X4(CAM4DV1_PTR cam, MATRIX4X4_PTR m)
{
// this function builds up a perspective to screen transformation
// matrix, the function assumes that you want to perform the
// transform in homogeneous coordinates and at raster time there will be
// a 4D->3D homogenous conversion and of course only the x,y points
// will be considered for the 2D rendering, thus you would use this
// function's matrix as if your perspective coordinates were still
// in homogeneous form when this matrix was applied, additionally
// the point of this matrix to scale and translate the perspective
// coordinates to screen coordinates, thus the matrix is built up
// assuming that the perspective coordinates are in normalized form for
// a 2x2 viewplane, that is, x: -1 to 1, y:-1/ar to 1/ar

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);
```


Playrix Ex. 1023, page 728

10 0672318350 CH07 05•02•2003 2:42 PM Page 704


```
Mat_Init_4X4(m, alpha,  0,   0,  0,
        0,   -beta, 0,  0,
        alpha,  beta, 1,  0,
        0,    0,   0,  1);

} // end Build_Perspective_To_Screen_4D_MATRIX4X4()
```

So now we make a call to build the matrix and then transform the object like this:

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D_MATRIX4X4 (&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mscr, TRANSFORM_TRANS_ONLY,1);
```

We almost have screen coordinates, but need to de-homogenize them with a call to

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

TIP

I know this stuff can be confusing, but the bottom line to remember is that we CANNOT perform division by z using a matrix, so we need to transfer z into the w component and then divide by it to obtain perspective transformations or non-affine transforms. That's the whole problem with 4D components. They're nice because you can use 4x4 matrices, but in most cases, I like to force w = 1 at all times and perform the perspective transform manually and divide by z myself, rather than having a separate de-homogenizing step where everything is divided by w.

Alright, that's the hard way. But let's assume that the vertices that make up the object have already been converted from 4D to 3D and are ready to go. So, all we need is a matrix version of the viewport transform that does the same math as the functional version we saw in `Perspective_To_Screen_OBJECT4DV1()`; that is, it assumes 3D coordinates and w = 1 already. Here's the matrix function that builds that matrix for us:

```
void Build_Perspective_To_Screen_MATRIX4X4(CAM4DV1_PTR cam, MATRIX4X4_PTR m)
{
// this function builds up a perspective to screen transformation
// matrix, the function assumes that you want to perform the
// transform in 2D/3D coordinates, that is, you have already converted
// the perspective coordinates from homogenous 4D to 3D before applying
// this matrix, additionally
// the point of this matrix is to scale and translate the perspective
// coordinates to screen coordinates, thus the matrix is built up
// assuming that the perspective coordinates are in normalized form for
// a 2x2 viewplane, that is, x: -1 to 1, y:-1/ar to 1/ar
// the only difference between this function and the version that
// assumes the coordinates are still in homogenous format is the
```

Playrix Ex. 1023, page 729

10 0672318350 CH07 05•02•2003 2:42 PM Page 705


```
// last column doesn't force w=z, in fact the z, and w results
// are irrelevent since we assume that BEFORE this matrix is applied
// all points are already converted from 4D->3D

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

Mat_Init_4X4(m, alpha,  0,   0,  0,
        0,   -beta, 0,  0,
        alpha,  beta, 1,  0,
        0,    0,   0,  1);

} // end Build_Perspective_To_Screen_MATRIX4X4()
```

With this function, we assume that the vertices are already in 3D form (technically 2D, and in perspective normalized coordinates, to boot). To use it, you would do this:

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mscr, TRANSFORM_TRANS_ONLY,1);
```

The point of all these examples is to show you that once you convert from 4D to 3D (usually in the perspective transform), the pipeline math and matrices slightly change and you have to have two branches. Personally, I prefer to transform from 4D to 3D perspective coordinates during the perspective transformation.

Viewport Transformation for Render Lists

The viewport transformation for render lists is identical to that of the objects, other than the data structure access. First, let's look at the functional version that assumes there was a previous normalized perspective transform with a divide by z (or w, however it was done) and the perspective coordinates just need to be scaled and translated to screen coordinates. Here's that function:

```
void Perspective_To_Screen_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                        CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms the perspective coordinates of the render
// list into screen coordinates, based on the sent viewport in the camera
// assuming that the viewplane coordinates were normalized
// you would use this function instead of the object based function
// if you decided earlier in the pipeline to turn each object into
// a list of polygons and then add them to the global render list
// you would only call this function if you previously performed
// a normalized perspective transform
```

Playrix Ex. 1023, page 730

10 0672318350 CH07 05•02•2003 2:42 PM Page 706


```
// transform each polygon in the render list from perspective to screen
// coordinates assumes the render list has already been transformed
// to normalized perspective coordinates and the result is in tvlist[]
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// acquire current polygon
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// is this polygon valid?
// transform this polygon if and only if it's not clipped, not culled,
// active, and visible, note however the concept of "backface" is
// irrelevant in a wire frame engine though
if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// all good, let's transform
for (int vertex = 0; vertex < 3; vertex++)
  {
  // the vertex is in perspective normalized coords from -1 to 1
  // on each axis, simple scale them and invert y axis and project
  // to screen

  // transform the vertex by the view parameters in the camera
  curr_poly->tvlist[vertex].x = alpha + alpha*curr_poly->tvlist[vertex].x;
  curr_poly->tvlist[vertex].y = beta - beta *curr_poly->tvlist[vertex].y;
  } // end for vertex

} // end for poly

} // end Perspective_To_Screen_RENDERLIST4DV1
```

Nothing more than a scale and y-axis inversion, as we would expect. Here's an example of the function being used after a call to perform the camera-to-perspective conversion that results in non-homogeneous coordinates:

```
// do the camera to perspective and divide by z
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// now map perspective cords to screen
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);
```

Playrix Ex. 1023, page 731

10 0672318350 CH07 05•02•2003 2:42 PM Page 707

At this point, I think you get the idea of the difference between 4D and 3D, so just as we did with objects, we can perform the transform with matrices. But we need to be careful to take into consideration of whether the perspective coordinates are in 4D or 3D. Here's the example of the coordinates staying in 4D form all way down:

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D(&cam, &mscr);

Transform_RENDERLIST4DV1(&rend_list, &mscr, TRANSFORM_TRANS_ONLY);
```

We almost have screen coordinates, but need to de-homogenize them with a call to

```
Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);
```

And finally, if the coordinates were already in 3D form, we can forget about the de-homogenizing phase and use the simpler perspective to screen transform that we've already seen. It builds a matrix, assuming 3D coordinates already, as shown here:

```
MATRIX4X4 mscr;

Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);

Transform_RENDERLIST4DV1(&rend_list, &mscr, TRANSFORM_TRANS_ONLY);
```

As you can see, the cool thing about matrices is that we can use them for anything. In this case, the only difference between doing things with objects and render lists is the final call to `Transform_*()`, which performs the transformation.

TIP

In real life, I would write the `Transform_*()` functions as virtual (and all this stuff as classes) so that I could call a single function and it would transform whatever the underlying object was, but I have to leave something for you to do later!

Hybrid Perspective Screen Transforms

Now it's time to use some common sense and do a little optimizing. I want to show you all the ways to do things, so you have no doubt about how things work. But unless you're completely asleep, you should realize by now that matrices are pretty, but definitely not friendly to the perspective transform. Thus, performing the perspective transform is easier to do manually. Moreover, having separate functions to perform the perspective and then the screen mapping makes sense in a lot of cases. But if you just want to get the darn image into a buffer, there's no need for a second function call because all we're really doing is a second scale and translation after the perspective transform. Hence, it's trivial to put these two operations into a single function and create a single transform function that

Playrix Ex. 1023, page 732

10 0672318350 CH07 05•02•2003 2:42 PM Page 708

does both the perspective to viewplane, and then the viewplane to viewport/screen transform.

The Camera-to-Screen Transform for Objects

Basically, the fastest way to perform the camera-to-screen transform is with a function that manually does it and divides by z itself. Here it is for objects:

```
void Camera_To_Perspective_Screen_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms the camera coordinates of an object
// into Screen scaled perspective coordinates, based on the
// sent camera object, that is, view_dist_h and view_dist_v
// should be set to cause the desired (width X height)
// projection of the vertices, but the function totally
// disregards the polygons themselves,
// it only works on the vertices in the vlist_trans[] list
// this is one way to do it, you might instead transform
// the global list of polygons in the render list since you
// are guaranteed that those polys represent geometry that
// has passed thru backfaces culling (if any)
// finally this function is really for experimental reasons only
// you would probably never let an object stay intact this far down
// the pipeline, since it's probably that there's only a single polygon
// that is visible! But this function has to transform the whole mesh!
// finally, the function also inverts the y axis, so the coordinates
// generated from this function ARE screen coordinates and ready for
// rendering

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// transform each vertex in the object to perspective screen coordinates
// assumes the object has already been transformed to camera
// coordinates and the result is in vlist_trans[]
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  float z = obj->vlist_trans[vertex].z;

  // transform the vertex by the view parameters in the camera
  obj->vlist_trans[vertex].x = cam->view_dist_h *
                obj->vlist_trans[vertex].x/z;
  obj->vlist_trans[vertex].y = cam->view_dist_v *
```

Playrix Ex. 1023, page 733

10 0672318350 CH07 05•02•2003 2:42 PM Page 709


```
                obj->vlist_trans[vertex].y *
                cam->aspect_ratio/z;
  // z = z, so no change

  // not that we are NOT dividing by the homogenous w coordinate since
  // we are not using a matrix operation for this version of the function

  // now the coordinates are in the range
  // x:(-viewport_width/2 to viewport_width/2)
  // and y:(-viewport_height/2 to viewport_height/2),
  // thus we need a translation and
  // since the Y-axis is inverted, we need to invert y
  // to complete the screen transform:
  obj->vlist_trans[vertex].x = obj->vlist_trans[vertex].x + alpha;
  obj->vlist_trans[vertex].y = -obj->vlist_trans[vertex].y + beta;

  } // end for vertex

} // end Camera_To_Perspective_Screen_OBJECT4DV1
```

Notice that we save even more math in the function. We don't have to scale the viewplane coordinates to the viewport because in the transform itself we project them onto a viewplane that's `viewport_width` × `viewport_height`, which saves a little more. Anyway, after this function, the coordinates in `vlist_trans[]` are totally ready to be drawn. Here's a sample call:

```
Camera_To_Perspective_Screen_OBJECT4DV1(&obj,&cam);
```

I have no idea why I keep giving examples of these incredibly trivial function calls, but it can't hurt! In any case, that's all there is to it! Of course, before the call, the coordinates better be in camera coordinates, but other than that, you're good to go.

The Camera-to-Screen Transform for Render Lists

There's nothing magic about performing the camera-to-screen transform at once for render lists either. It's just a slightly different data structure access. Here's the function:

```
void Camera_To_Perspective_Screen_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                         CAM4DV1_PTR cam)
{
// NOTE: this is not a matrix based function
// this function transforms the camera coordinates of an object
// into Screen scaled perspective coordinates, based on the
// sent camera object, that is, view_dist_h and view_dist_v
// should be set to cause the desired (viewport_width X viewport_height)
```

Playrix Ex. 1023, page 734

10 0672318350 CH07 05•02•2003 2:42 PM Page 710


```
// it only works on the vertices in the tvlist[] list
// finally, the function also inverts the y axis, so the coordinates
// generated from this function ARE screen coordinates and ready for
// rendering

// transform each polygon in the render list to perspective screen
// coordinates assumes the render list has already been transformed
// to camera coordinates and the result is in tvlist[]
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// acquire current polygon
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// is this polygon valid?
// transform this polygon if and only if it's not clipped, not culled,
// active, and visible, note however the concept of "backface" is
// irrelevant in a wire frame engine though
if ((curr_poly==NULL) ¦¦ !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ¦¦
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ¦¦
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// all good, let's transform
for (int vertex = 0; vertex < 3; vertex++)
  {
  float z = curr_poly->tvlist[vertex].z;

  // transform the vertex by the view parameters in the camera
  curr_poly->tvlist[vertex].x = cam->view_dist_h *
                curr_poly->tvlist[vertex].x/z;
  curr_poly->tvlist[vertex].y = cam->view_dist_v *
                curr_poly->tvlist[vertex].y *
                cam->aspect_ratio/z;
  // z = z, so no change

  // not that we are NOT dividing by the homogenous w coordinate since
  // we are not using a matrix operation for this version of the function

  // now the coordinates are in the range
  // x:(-viewport_width/2 to viewport_width/2)
```

Playrix Ex. 1023, page 735

10 0672318350 CH07 05•02•2003 2:42 PM Page 711


```
  // and y:(-viewport_height/2 to viewport_height/2),
  // thus we need a translation and
  // since the Y-axis is inverted, we need to invert y to complete the screen
  // transform:
  curr_poly->tvlist[vertex].x = curr_poly->tvlist[vertex].x + alpha;
  curr_poly->tvlist[vertex].y = -curr_poly->tvlist[vertex].y + beta;

  } // end for vertex

} // end for poly

} // end Camera_To_Perspective_Screen_RENDERLIST4DV1
```

And here's a call to it:

```
Camera_To_Perspective_Screen_RENDERLIST4DV1(&rend_list,
&cam);
```

Wow! That's it. We're done with the major parts of the 3D pipeline! Probably the only confusing part is that when we get to the perspective transform, we have to decide whether to do it manually or with a matrix, and then whether we want to hold on to the 4D coordinates. On the other hand, we can perform the perspective and screen transform all at once and save a lot of time and drama. We'll end up using this method most of the time, or at least using manual functions for the perspective and screen transform that converts to 3D coordinates (that is, de-homogenizes) because we don't want to be converted 4D to 3D late in the pipeline. However, as I said when using hardware, we may not have this option, and matrices and 4D coordinates may be what the hardware likes.

Rendering a 3D World

I never thought we would get here—thank `char s[] = {71, 79, 68, 0}`! Of course, my dilemma was that for the previous zillion pages of this chapter, I couldn't show you a single demo because we have to first understand and code each step of the pipeline. But now we're done and we can write all the demos we want—and we will! But first, let's take yet another look at the 3D pipeline and then we'll write some simple graphics functions that draw wireframe meshes that feed from either `OBJECT4DV1`s or `RENDERLIST4DV1`s.

The 3D Pipeline As It Stands

Currently, we have all the code we need to load 3D objects off disk and store them in an `OBJECT4DV1` structure and then push one or more objects all the way through the 3D pipeline. Or we can push one or more `OBJECT4DV1`s down the 3D pipeline as far as we wish, rip it apart into its constituent polygons, and then insert them into a master polygon list via the data structure `RENDERLIST4DV1`. How you want to do it is really up to you; both methods have strengths and weaknesses. But in general, most 3D engines use a

Playrix Ex. 1023, page 736

10 0672318350 CH07 05•02•2003 2:42 PM Page 712

hybrid of both. They work with objects for the things that move around, but the environment might be a giant mesh that's sectorized and simply inserted into the render list along with the objects at some point. But for now, we're keeping things simple and assuming that our games are made from objects, and that there's no interior or exterior environments we need to deal with.

With that in mind, Figure 7.19 shows the different path ways geometry can take, but in general, the following steps must take place:

Stage 0: Load objects and position them

Stage 1: Local-to-world transform

Stage 2: Object and back-face removal

Stage 3: World-to-camera transform

Stage 4: Camera-to-perspective transform

Stage 5: Perspective-to-screen transform

Stage 6: Render the geometry

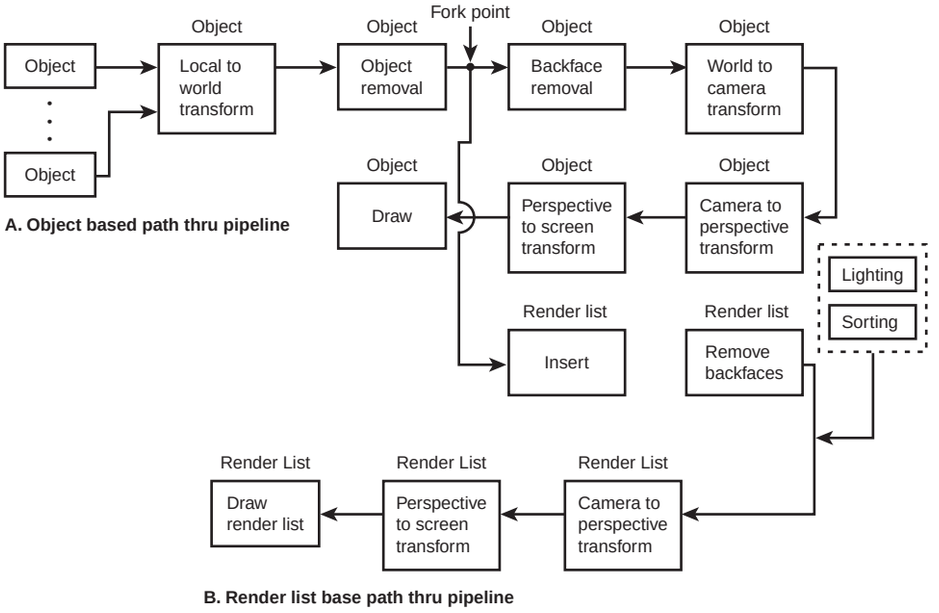


FIGURE 7.19 The viewing pipeline and multiple render paths.

Now, you might convert your objects into polygons anywhere after Stage 2 because you want to at least try to remove objects while they're intact. Once you convert them to

Playrix Ex. 1023, page 737

10 0672318350 CH07 05•02•2003 2:42 PM Page 713

polygons, you lose their structure and can't think of them as objects anymore. We've completely left out 3D clipping, but as long as we remove objects, we're somewhat safe at this point because we clip the objects to the near z-plane. For the most part, we won't get any geometry that has z = negative or z = 0 vertices, but we need to deal with this, and we will in the next chapter when we perform full 3D clipping. For now, we assume that all the projected vertices are reasonably safe, and if they *do* extend beyond the 2D viewport, they'll be clipped by the 2D drawing functions.

Okay, so no matter how you sequence our geometry down the 3D pipeline, at some point we're going to have one of two things: a list of `OBJECT4DV1`s that need to be rendered or a single `RENDERLIST4DV1` that represents all the polygons and needs to be rendered. Of course, you could have more than one render list, but for now, let's assume one that basically would contain all the objects. So, what we need are two functions: one that can draw all the polygons in an `OBJECT4DV1`, and one that can draw all the polygons in a `RENDERLIST4DV1`.

Drawing an `OBJECT4DV1` Before we begin, let's remember that we're going to use wireframe graphics without any hidden line removal or sorting. That means that you'll be able to see through the objects. Moreover, the objects may be drawn in the wrong order because we aren't taking into consideration their z-depth to sort them or use any other type of proper ordering, but who cares? They *are* hollow! Therefore, all we need to do is write a function that basically assumes that the vertex list in the transformed object's vertice list is valid and in screen coordinates, and just run through the geometry and draw each triangle. Here's the function that does the job. Note that the function tests to see if each polygon is visible, two-sided, can be clipped, or is a back-face.

```
void Draw_OBJECT4DV1_Wire16(OBJECT4DV1_PTR obj,
              UCHAR *video_buffer, int lpitch)

{
// this function renders an object to the screen in wireframe,
// 16 bit mode, it has no regard at all about hidden surface removal,
// etc. the function only exists as an easy way to render an object
// without converting it into polygons, the function assumes all
// coordinates are screen coordinates, but will perform 2D clipping

// iterate thru the poly list of the object and simply draw
// each polygon
for (int poly=0; poly < obj->num_polys; poly++)
  {
  // render this polygon if and only if it's not clipped, not culled,
  // active, and visible, note however the concecpt of "backface" is
  // irrelevant in a wire frame engine though
  if (!(obj->plist[poly].state & POLY4DV1_STATE_ACTIVE) ¦¦
     (obj->plist[poly].state & POLY4DV1_STATE_CLIPPED ) ¦¦
```


Playrix Ex. 1023, page 738

10 0672318350 CH07 05•02•2003 2:42 PM Page 714


```
    (obj->plist[poly].state & POLY4DV1_STATE_BACKFACE) )
   continue; // move onto next poly

  // extract vertex indices into master list, rember the polygons are
  // NOT self contained, but based on the vertex list stored in the object
  // itself
  int vindex_0 = obj->plist[poly].vert[0];
  int vindex_1 = obj->plist[poly].vert[1];
  int vindex_2 = obj->plist[poly].vert[2];

  // draw the lines now
  Draw_Clip_Line16(obj->vlist_trans[ vindex_0 ].x,
           obj->vlist_trans[ vindex_0 ].y,
           obj->vlist_trans[ vindex_1 ].x,
           obj->vlist_trans[ vindex_1 ].y,
           obj->plist[poly].color,
           video_buffer, lpitch);

  Draw_Clip_Line16(obj->vlist_trans[ vindex_1 ].x,
           obj->vlist_trans[ vindex_1 ].y,
           obj->vlist_trans[ vindex_2 ].x,
           obj->vlist_trans[ vindex_2 ].y,
           obj->plist[poly].color,
           video_buffer, lpitch);

  Draw_Clip_Line16(obj->vlist_trans[ vindex_2 ].x,
           obj->vlist_trans[ vindex_2 ].y,
           obj->vlist_trans[ vindex_0 ].x,
           obj->vlist_trans[ vindex_0 ].y,
           obj->plist[poly].color,
           video_buffer, lpitch);

  } // end for poly

} // end Draw_OBJECT4DV1_Wire
```

The function is beautiful, huh? With our virtual computer system and graphics buffer, we can write 3D functions all day and not know a thing about the hardware—all we need is the address of the video buffer and the memory pitch. Anyway, the function makes use of a couple of the 2D functions for drawing clipped lines and does nothing more than traverse the polygon list and draw all the triangles that make up the 3D object. Here's a sample call where you would have already set up the graphics system, performed the 3D pipeline, and `obj` was ready to go:

```
Draw_OBJECT4DV1_Wire16(&obj, video_buffer,lpitch);
```

Playrix Ex. 1023, page 739

10 0672318350 CH07 05•02•2003 2:42 PM Page 715

Drawing a `RENDERLIST4DV1` The drawing function for the render lists is identical to the objects, but with different data structure access. Here it is:

```
void Draw_RENDERLIST4DV1_Wire16(RENDERLIST4DV1_PTR rend_list,
                UCHAR *video_buffer, int lpitch)
{
// this function "executes" the render list or in other words
// draws all the faces in the list in wire frame 16bit mode
// note there is no need to sort wire frame polygons, but
// later we will need to, so hidden surfaces stay hidden
// also, we leave it to the function to determine the bitdepth
// and call the correct rasterizer

// at this point, all we have is a list of polygons and it's time
// to draw them
for (int poly=0; poly < rend_list->num_polys; poly++)
  {
  // render this polygon if and only if it's not clipped, not culled,
  // active, and visible, note however the concecpt of "backface" is
  // irrelevant in a wire frame engine though
  if (!(rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_ACTIVE) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_CLIPPED ) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly


  // draw the triangle edge one, note that clipping was already set up
  // by 2D initialization, so line clipper will clip all polys out
  // of the 2D screen/window boundary
  Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);

  Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);
```

Playrix Ex. 1023, page 740

10 0672318350 CH07 05•02•2003 2:42 PM Page 716


```
  Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[2].x,
        rend_list->poly_ptrs[poly]->tvlist[2].y,
        rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->color,
        video_buffer, lpitch);

  } // end for poly

} // end Draw_RENDERLIST4DV1_Wire
```

Calling it is no different, we just need a render list rather than an object:

```
Draw_RENDERLIST4DV1_Wire16(&rend_list, video_buffer,lpitch);
```

That's it, my large-brained friend! Now we're ready to see some demos. Now that we have the pipeline and rendering algorithms all done, I would like to write a number of demos to review all the different techniques we covered in the previous sections. So, the next section is going to run through a number of examples to give you a good idea how to do everything.

3D Demos

Finally, some full demos. It's been torture, hasn't it? Now, don't get too built up, we're not going to do anything other than basic wireframe examples based on all the functions we've built. But as I said, after this chapter, you could easily write *Battle Zone*, or something of that nature. Each demo will have a title and set of specifications that generally describe what's going on, so you can jump around this section of the chapter if you want and find something you're interested in. Additionally, I'm not going to show you the entire program for each demo because you've already seen all the functions themselves. Rather, just the important stuff in the main loop will be listed. To compile any of these demos, you'll need the following files in addition to the main .CPP file of the demo:

```
T3DLIB1.CPP¦H

T3DLIB2.CPP¦H

T3DLIB3.CPP¦H

T3DLIB4.CPP¦H

T3DLIB5.CPP¦H

DDRAW.LIB

DSOUND.LIB
```

Playrix Ex. 1023, page 741

10 0672318350 CH07 05•02•2003 2:42 PM Page 717

DINPUT.LIB

DINPUT8.LIB

And all the .PLG files that are located in this chapter's directory, of course.

Single 3D Triangle—DEMOII7_1.CPP¦EXE

Object(s) Loaded: None; manual creation of single polygon

Type of Camera: Euler, fixed position, 90 degree FOV normalized viewplane

Type of Projection: Perspective followed by viewport mapping

Math Used for 3D Pipeline: All manual function calls, no transforms based on individual matrices

Back-face Removal: None

Object Removal: None

Type of Rendering Geometry: Render list with single polygon

DEMOII7_1.CPP¦EXE is as basic as it gets. The program creates a single polygon, rotates it around the +y-axis, and that's it! Here are the important globals for the demo:

```
// initialize camera position and direction
POINT4D cam_pos = {0,0,-100,1};
VECTOR4D cam_dir = {0,0,0,1};

// all your initialization code goes here...
VECTOR4D vscale={.5,.5,.5,1}, vpos = {0,0,0,1}, vrot = {0,0,0,1};

CAM4DV1    cam;           // the single camera
RENDERLIST4DV1 rend_list;        // the single renderlist
POLYF4DV1   poly1;          // our lonely polygon
POINT4D    poly1_pos = {0,0,100,1}; // world position of polygon
```

And here's the 3D code in the Game_Init() function:

```
// initialize math engine
Build_Sin_Cos_Tables();

// initialize a single polygon
poly1.state = POLY4DV1_STATE_ACTIVE;
poly1.attr  = 0;
poly1.color = RGB16Bit(0,255,0);

poly1.vlist[0].x = 0;
poly1.vlist[0].y = 50;
```

Playrix Ex. 1023, page 742

10 0672318350 CH07 05•02•2003 2:42 PM Page 718


```
poly1.vlist[0].z = 0;
poly1.vlist[0].w = 1;

poly1.vlist[1].x = 50;
poly1.vlist[1].y = -50;
poly1.vlist[1].z = 0;
poly1.vlist[1].w = 1;

poly1.vlist[2].x = -50;
poly1.vlist[2].y = -50;
poly1.vlist[2].z = 0;
poly1.vlist[2].w = 1;

poly1.next = poly1.prev = NULL;

// initialize the camera with 90 FOV, normalized coordinates
Init_CAM4DV1(&cam,   // the camera object
       &cam_pos, // initial camera position
       &cam_dir, // initial camera angles
       50.0,   // near and far clipping planes
       500.0,
       90.0,   // field of view in degrees
       1.0,    // viewing distance
       0,     // size of viewplane-1
       0,
       WINDOW_WIDTH,  // size of final screen viewport
       WINDOW_HEIGHT);
```

Finally, here's the main core of the `Game_Main()` function that performs the entire 3D pipeline and rendering of the triangle:

```
int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for you game go here!

static MATRIX4X4 mrot; // general rotation matrix
static float ang_y = 0;   // rotation angle

int index; // looping var

// start the timing clock
Start_Clock();
```

Playrix Ex. 1023, page 743

10 0672318350 CH07 05•02•2003 2:42 PM Page 719


```
// clear the drawing surface
DDraw_Fill_Surface(lpddsback, 0);

// read keyboard and other devices here
DInput_Read_Keyboard();

// game logic here...

// initialize the renderlist
Reset_RENDERLIST4DV1(&rend_list);

// insert polygon into the renderlist
Insert_POLYF4DV1_RENDERLIST4DV1(&rend_list, &poly1);

// generate rotation matrix around y axis
Build_XYZ_Rotation_MATRIX4X4(0, ang_y, 0, &mrot);

// rotate polygon slowly
if (++ang_y >= 360.0) ang_y = 0;

// rotate the local coords of single polygon in renderlist
Transform_RENDERLIST4DV1(&rend_list, &mrot, TRANSFORM_LOCAL_ONLY);

// perform local/model to world transform
Model_To_World_RENDERLIST4DV1(&rend_list, &poly1_pos);

// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// apply world to camera transform
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// apply camera to perspective transformation
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// apply screen transform
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// lock the back buffer
DDraw_Lock_Back_Surface();

// render the polygon list
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer, back_lpitch);
```


Playrix Ex. 1023, page 744

10 0672318350 CH07 05•02•2003 2:42 PM Page 720


```
// unlock the back buffer
DDraw_Unlock_Back_Surface();

// flip the surfaces
DDraw_Flip();

// sync to 30ish fps
Wait_Clock(30);

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) ¦¦ keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);
  } // end if

// return success
return(1);

} // end Game_Main
```

Figure 7.20 is a screenshot of DEMOII7_1.EXE in action. Very exciting, as you can see! The main point of this demo is to show that with a dozen or so calls, we can create a full 3D wireframe display—that's nothing to laugh at.



FIGURE 7.20 Screenshot of DEMOII7_1.EXE.

3D Wireframe Cube—DEMOII7_2.CPP¦EXE

Object(s) Loaded: Simple Cube (CUBE1.PLG)

Type of Camera: Euler, fixed position, 90 degree FOV normalized viewplane of 2x2 with viewport of 400x400

Playrix Ex. 1023, page 745

10 0672318350 CH07 05•02•2003 2:42 PM Page 721

Type of Projection: Perspective followed by viewport mapping

Math Used for 3D Pipeline: All manual function calls; no transforms based on individual matrices

Back-face Removal: None

Object Removal: None

Type of Rendering Geometry: Object rendering, no render list

Now that you've see a single polygon rotating, let's do something a little more exciting, shall we? `DEMOII7_2.CPP¦EXE` is similar to the previous demo, but this time we're going to use our mesh-loading software to load in the cube mesh `CUBE1.PLG` shown here:

```
# begin plg/plx file

# simple cube
tri 8 12

# vertex list
 5 5 5
-5 5 5
-5 5 -5
 5 5 -5
 5 -5 5
-5 -5 5
-5 -5 -5
 5 -5 -5


# polygon list
0xd0f0 3 2 1 0
0xd0f0 3 3 2 0
0xd0f0 3 4 7 0
0xd0f0 3 7 3 0
0xd0f0 3 6 7 4
0xd0f0 3 5 6 4
0xd0f0 3 2 6 1
0xd0f0 3 6 5 1
0xd0f0 3 7 6 3
0xd0f0 3 6 2 3
0xd0f0 3 5 4 0
0xd0f0 3 1 5 0
```


Playrix Ex. 1023, page 746

10 0672318350 CH07 05•02•2003 2:42 PM Page 722

Basically, it's a 10x10x10 cube. After the mesh is loaded as an `OBJECT4DV1`, it's slowly rotated in the main loop and transformed down the 3D pipeline. However, we will NOT convert the cube into polygons and insert them into a render list; we'll leave the cube as an object so that you can see that it's possible to keep an object intact all the way down the pipeline.

Here are the important globals for the demo:

```
// initialize camera position and direction
POINT4D cam_pos = {0,0,0,1};
VECTOR4D cam_dir = {0,0,0,1};

// all your initialization code goes here...
VECTOR4D vscale={5.0,5.0,5.0,1}, vpos = {0,0,0,1}, vrot = {0,0,0,1};

CAM4DV1  cam;   // the single camera
OBJECT4DV1 obj;   // used to hold our cube mesh
```

Notice the value of `vscale`. Remember that this vector scales the mesh during loading; hence, the 10x10x10 cube will be scaled during loading to 50x50x50. Of course, we could have just changed the model itself, but that's the point of the flexibility in the loading function: to use the same model and transform it during loading to allow slight morphing of the mesh.

Now, let's look at the 3D part of the initialization code in `Game_Init()`. It's the same as before, but you'll see the object loaded:

```
// initialize the camera with 90 FOV, normalized coordinates
Init_CAM4DV1(&cam,   // the camera object
       &cam_pos, // initial camera position
       &cam_dir, // initial camera angles
       50.0,   // near and far clipping planes
       500.0,
       90.0,   // field of view in degrees
       WINDOW_WIDTH,  // size of final screen viewport
       WINDOW_HEIGHT);

// load the cube
Load_OBJECT4DV1_PLG(&obj, "cube1.plg",&vscale, &vpos, &vrot);

// set the position of the cube in the world
obj.world_pos.x = 0;
obj.world_pos.y = 0;
obj.world_pos.z = 100;
```

Playrix Ex. 1023, page 747

10 0672318350 CH07 05•02•2003 2:42 PM Page 723

Note that the object is positioned 100 units from the origin of the world. This is so it looks really big and we get a lot of perspective. Finally, here's the `Game_Main()` so that you can see the pipeline calls:

```
int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for you game go here!

static MATRIX4X4 mrot; // general rotation matrix

int index; // looping var

// start the timing clock
Start_Clock();

// clear the drawing surface
DDraw_Fill_Surface(lpddsback, 0);

// read keyboard and other devices here
DInput_Read_Keyboard();

// game logic here...

// reset the object (this only matters for backface and object removal)
Reset_OBJECT4DV1(&obj);

// generate rotation matrix around y axis
Build_XYZ_Rotation_MATRIX4X4(0, 5, 0, &mrot);

// rotate the local coords of single polygon in renderlist
Transform_OBJECT4DV1(&obj, &mrot, TRANSFORM_LOCAL_ONLY,1);

// perform local/model to world transform
Model_To_World_OBJECT4DV1(&obj);

// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```


Playrix Ex. 1023, page 748

10 0672318350 CH07 05•02•2003 2:42 PM Page 724


```
// apply world to camera transform
World_To_Camera_OBJECT4DV1(&obj, &cam);

// apply camera to perspective transformation
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);

// apply screen transform
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);

// lock the back buffer
DDraw_Lock_Back_Surface();

// render the object
Draw_OBJECT4DV1_Wire16(&obj, back_buffer, back_lpitch);

// unlock the back buffer
DDraw_Unlock_Back_Surface();

// flip the surfaces
DDraw_Flip();

// sync to 30ish fps
Wait_Clock(30);

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) ¦¦ keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if

// return success
return(1);

} // end Game_Main
```

Again, only a dozen or so important calls and we have an entire 3D demo up and running! Figure 7.21 is a screenshot of `DEMOII7_2.EXE` in action. Notice the amount of perspective. This is due to the field of view and the proximity of the object to the viewpoint.

Playrix Ex. 1023, page 749

10 0672318350 CH07 05•02•2003 2:42 PM Page 725



FIGURE 7.21 Screenshot of DEMOII7_2.EXE.

3D Wireframe Cube with Back-face Removal—DEMOII7_3.CPP¦EXE

Object(s) Loaded: Simple cube (CUBE2.PLG)

Type of Camera: Euler, fixed position, 90 degree FOV normalized viewplane of -1/ar to +1/ar with viewport of 640x480

Type of Projection: Perspective followed by viewport mapping

Math Used for 3D Pipeline: All manual function calls; no transforms based on individual matrices

Back-face Removal: Enabled

Object Removal: None

Type of Rendering Geometry: Object rendering; no render list

DEMOII7_3.CPP¦EXE is identical to DEMOII7_2.CPP¦EXE except that back-face removal has been enabled along with a larger viewport. Additionally, to make back-face removal work, we need to take a couple things into consideration. The first is in the model mesh itself. Remember that when loading a mesh, a polygon can be one- or two-sided. If it's one-sided, back-face removal makes sense. But if it's two-sided, it doesn't make sense because we can see the polygon from both sides. Now, in wireframe mode, it's obviously less important to have back-face removal, but we can still enable it in the mesh data if we like. The second thing we have to do is, of course, test for a back-face during the 3D pipeline somewhere after the world transform, but before the camera transform.

First, let's check to see whether back-faces were enabled in the CUBE1.PLG mesh data. Here's one of the polygon descriptor lines from the file:

```
# polygon list
0xd0f0 3 2 1 0 # polygon 0
```

Playrix Ex. 1023, page 750

10 0672318350 CH07 05•02•2003 2:42 PM Page 726

The first number is the polygon descriptor, and if you recall from the code in the previous section on loading .PLG/.PLX files, the D bit in the 16-bit surface descriptor defines whether the polygon is two-sided or not:

```
CSSD ¦ RRRR¦ GGGG ¦ BBBB
```

In other words, the D bit is the lower bit of the leftmost hex digit. In the case of this surface descriptor, the digit is hex 0xD, which is 1101b in binary; therefore, the two-sided flag is on! We must disable it. To do so, we'll need another .PLG file, so I've created one and called it `CUBE2.PLG`. It's on the CD and has this bit reset, so the descriptor looks like 1100b, or 0xC in hex. The analogous polygon descriptor in the preceding example would now be

```
# polygon list
0xc0f0 3 2 1 0 # polygon 0
```

To remove the back-faces, a call is made to

```
// remove backfaces
Remove_Backfaces_OBJECT4DV1(&obj, &cam);
```

This call is made after the camera matrix is created for the viewing angles, but before the general world-to-camera transformation takes place. The call is placed at this particular point so that polygons that are not visible aren't transformed during the world-to-camera transformation.

Figure 7.22 is a screenshot of `DEMOII7_3.EXE` in action. Try pressing the up and down arrow keys and watch the cube rotate around the x-axis.

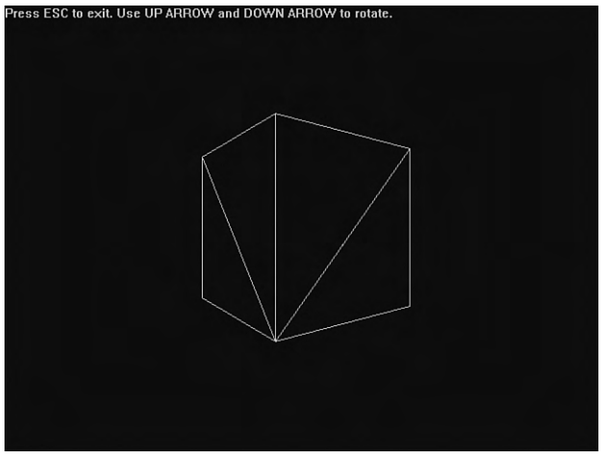


FIGURE 7.22 Screenshot of `DEMOII7_3.EXE`.

Playrix Ex. 1023, page 751

10 0672318350 CH07 05•02•2003 2:42 PM Page 727

3D Tank Demo—DEMOII7_4.CPP¦EXE

Object(s) Loaded: Simple tank (TANK1.PLG)

Type of Camera: Euler, rotatable, 90 degree FOV normalized viewplane of -1/ar to +1/ar with viewport of 400x400

Type of Projection: Perspective followed by viewport mapping

Math Used for 3D Pipeline: All manual function calls; no transforms based on individual matrices

Back-face Removal: Disabled

Object Removal: Yes

Type of Rendering Geometry: Render list with multiple objects inserted

Figure 7.23 depicts the hard-hitting 3D gaming action of DEMOII7_4.CPP! Well, at least there's some tanks in there <BG>. Anyway, this is a fairly complex demo. The camera is set up as usual, the tank object is loaded, and the demo proceeds into the main loop.



FIGURE 7.23 Screenshot of DEMOII7_4.EXE.

The interesting thing about this demo is how I created multiple copies of the tank: Instead of loading multiple tanks, I simply changed the position of the master tank and then inserted it into the master rendering list multiple times. Hence, the tank's polygons at a particular world position are transformed and then inserted into the rendering list, making it seem like there are multiple tanks. When the world-to-camera and perspective transformations are performed on the rendering list, we have multiple tanks! Here's an excerpt from the program that demonstrates the important parts of the main loop; I've bolded the copy/insertion code:

```
int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
```

Playrix Ex. 1023, page 752

10 0672318350 CH07 05•02•2003 2:42 PM Page 728


```
// continuously in real-time this is like main() in C
// all the calls for you game go here!

static MATRIX4X4 mrot; // general rotation matrix
static float x_ang = 0, y_ang = 5, z_ang = 0;
char work_string[256];

int index; // looping var

// start the timing clock
Start_Clock();

// clear the drawing surface
DDraw_Fill_Surface(lpddsback, 0);

// read keyboard and other devices here
DInput_Read_Keyboard();

// game logic here...

// reset the render list
Reset_RENDERLIST4DV1(&rend_list);

// reset angles
x_ang = 0;
y_ang = 1;
z_ang = 0;

// is user trying to rotate camera
if (KEY_DOWN(VK_DOWN))
  cam.dir.x+=1;
else
if (KEY_DOWN(VK_UP))
  cam.dir.x-=1;

// is user trying to rotate camera
if (KEY_DOWN(VK_RIGHT))
  cam.dir.y-=1;
else
if (KEY_DOWN(VK_LEFT))
  cam.dir.y+=1;

// generate rotation matrix around y axis
Build_XYZ_Rotation_MATRIX4X4(x_ang, y_ang, z_ang, &mrot);
```

Playrix Ex. 1023, page 753

10 0672318350 CH07 05•02•2003 2:42 PM Page 729


```
// rotate the local coords of the object
Transform_OBJECT4DV1(&obj, &mrot, TRANSFORM_LOCAL_ONLY,1);

// now cull the current object
strcpy(buffer,"Objects Culled: ");

for (int x=-NUM_OBJECTS/2; x < NUM_OBJECTS/2; x++)
  for (int z=-NUM_OBJECTS/2; z < NUM_OBJECTS/2; z++)
  {
  // reset the object (this only matters for backface and object removal)
  Reset_OBJECT4DV1(&obj);

  // set position of object
  obj.world_pos.x = x*OBJECT_SPACING+OBJECT_SPACING/2;
  obj.world_pos.y = 0;
  obj.world_pos.z = 500+z*OBJECT_SPACING+OBJECT_SPACING/2;

  // attempt to cull object
  if (!Cull_OBJECT4DV1(&obj, &cam, CULL_OBJECT_XYZ_PLANES))
    {
    // if we get here then the object is visible at this world position
    // so we can insert it into the rendering list
    // perform local/model to world transform
    Model_To_World_OBJECT4DV1(&obj);

    // insert the object into render list
    Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj);
    } // end if
  else
    {
    sprintf(work_string, "[%d, %d] ", x,z);
    strcat(buffer, work_string);
    }

  } // end for

Draw_Text_GDI(buffer, 0, WINDOW_HEIGHT-20, RGB(0,255,0), lpddsback);

// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// remove backfaces
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);
```


Playrix Ex. 1023, page 754

10 0672318350 CH07 05•02•2003 2:42 PM Page 730


```
// apply world to camera transform
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// apply camera to perspective transformation
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// apply screen transform
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// draw instructions
Draw_Text_GDI("Press ESC to exit. Use ARROW keys to rotate camera.",
       0, 0, RGB(0,255,0), lpddsback);

// lock the back buffer
DDraw_Lock_Back_Surface();

// render the object
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer, back_lpitch);

// unlock the back buffer
DDraw_Unlock_Back_Surface();

// flip the surfaces
DDraw_Flip();

// sync to 30ish fps
Wait_Clock(30);

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE) ¦¦ keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if

// return success
return(1);

} // end Game_Main
```

Now you see why having a rendering list is so important, and is a better way to approach rendering than rendering separate objects themselves. Having objects is fine for transformations, animation, AI, logic, and so on, but once the rendering phase starts, it's better to

Playrix Ex. 1023, page 755

10 0672318350 CH07 05•02•2003 2:42 PM Page 731

simply decompose objects into polygons (after object culling) and then insert the polygons into a master list.

3D Tanks Demo with Flying Camera—DEMOII7_5.CPP¦EXE

Object(s) Loaded: Simple tank (TANK1.PLG)

Type of Camera: UVN, rotatable, 90 degree FOV normalized viewplane of -1/ar to +1/ar with viewport of 800x600, full-screen

Type of Projection: Perspective followed by viewport mapping

Math Used for 3D Pipeline: All manual function calls; no transforms based on individual matrices

Back-face Removal: Disabled

Object Removal: Yes

Type of Rendering Geometry: Render list with multiple objects inserted

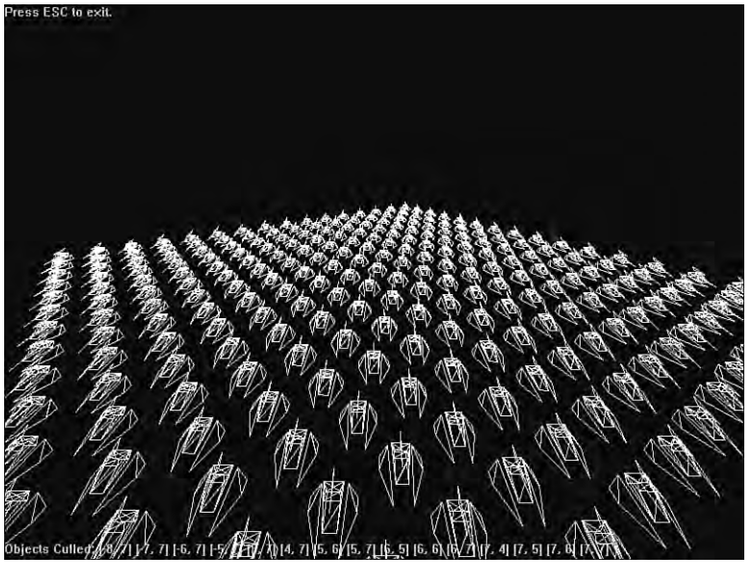


FIGURE 7.24 Screenshot of DEMOII7_5.EXE.

Figure 7.24 is a screenshot of the most complex demo yet. I was shooting for a clone-wars-attack kind of thing! Anyway, this demo uses the UVN camera model, so the only thing that's really different from the other demos is the call to Build_CAM4DV1_Matrix_UVN() rather than the Euler function call. Moreover, the UVN_MODE_SIMPLE parameter is used, which tells the camera function that we're supplying only the target position manually.

Playrix Ex. 1023, page 756

10 0672318350 CH07 05•02•2003 2:42 PM Page 732

Thus the function blindly computes the **u**,**v**,**n** vectors without further ado. The `Game_Main()` is very similar to the previous demo, but has a couple of interesting points: the computation of the camera position and, of course, the UVN camera model. Here's the code that positions the camera:

```
// compute new camera position, rides on a circular path
// with a sinuoid modulation
cam.pos.x = camera_distance*Fast_Cos(view_angle);
cam.pos.y = camera_distance*Fast_Sin(view_angle);
cam.pos.z = 2*camera_distance*Fast_Sin(view_angle);

// advance angle to circle around
if ((view_angle+=1)>=360)
  view_angle = 0;

// generate camera matrix
Build_CAM4DV1_Matrix_UVN(&cam, UVN_MODE_SIMPLE);
```

As you can see, there's not much to it! Figure 7.25 depicts the world setup and what's happening. This is a good example of very simple math creating a very interesting camera motion. In this case, I have a circular path for the camera (or elliptical, depending on the multiplication factors), along with bouncing the height of the camera above the x-z plane along a sine wave. The effect is a very nice camera motion. However, realize that this kind of camera control is possible only because we're using the UVN model. The UVN model gives the control we need in this case because it's well suited for the case in which you have a point of interest you want to "look at" and a camera position, which is the case in this demo. I position the camera on the circular ring surrounding the target, which is the center of the tank army, and that's it!

The interesting thing about this camera model is that it's great for viewing the action from a third-person perspective. That is, using the camera position and a target works when you want to look at some point of interest, and you want the math to figure out the UVN vectors, which in turn compute the rotation matrix. However, the second construction we developed for the UVN model (that is, where we give the camera position, but instead of a target we select the elevation and heading), works much better for a first-person game where you want to see through the character's eyes. Let's do a final demo using that technique and the spherical camera system plugged into the UVN model.

Battle Zone Walkthrough Demo—`DEMOII7_6.CPP¦EXE`

Object(s) Loaded: Tank (`TANK1.PLG`), Tower (`TOWER1.PLG`)

Type of Camera: Euler with heading, rotatable, 120 degree FOV normalized viewplane of -1/`ar` to +1/`ar` with viewport of 800x600, full-screen

Type of Projection: Perspective followed by viewport mapping

Playrix Ex. 1023, page 757

10 0672318350 CH07 05•02•2003 2:42 PM Page 733

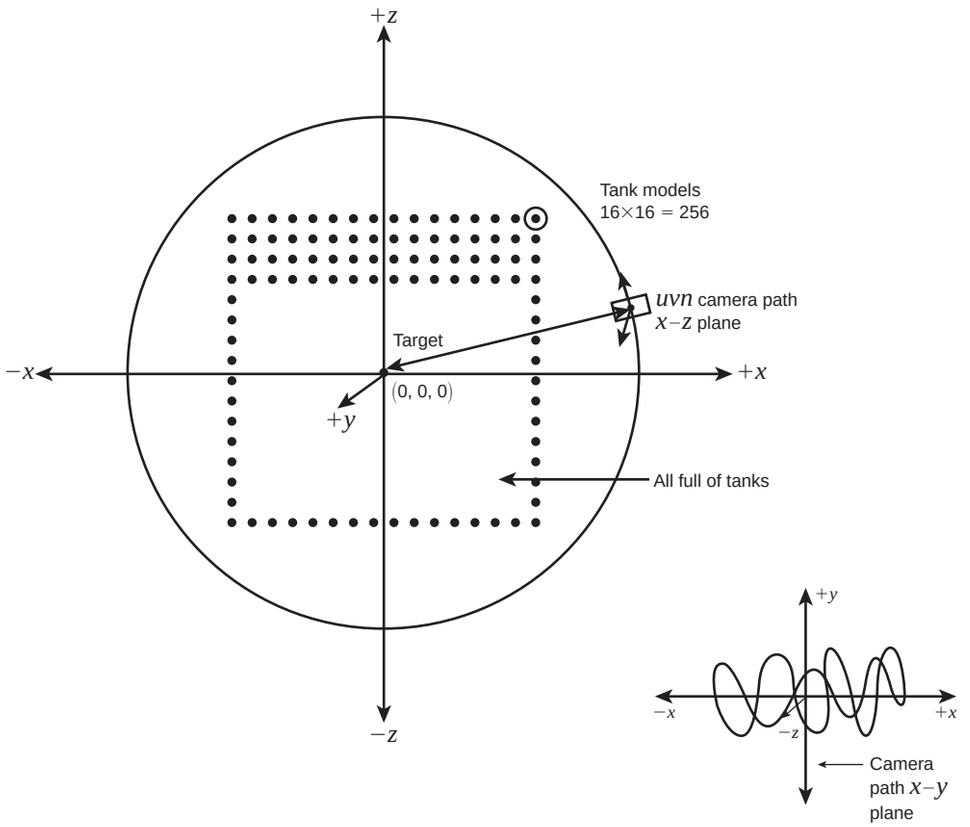


FIGURE 7.25 World setup for the demo.

Math Used for 3D Pipeline: All manual function calls; no transforms based on individual matrices

Back-face Removal: Disabled

Object Removal: Yes

Type of Rendering Geometry: Render list with multiple objects inserted

This is the coolest demo yet! Basically, the demo puts everything we have seen together and has a few game programming tricks thrown in. The demo consists of three object types: towers, tanks, and ground markers. Because we're still using wireframe rendering, it's difficult to create a feeling for depth, so I added a large number of regularly placed small pyramids all over the game arena to create a sense of a ground plane. Additionally, there's a sky and ground created with two blitter-filled rectangles. See Figure 7.26.

Playrix Ex. 1023, page 758

10 0672318350 CH07 05•02•2003 2:42 PM Page 734



FIGURE 7.26 Screenshot of `DEMOII7_6.EXE`.

The important parts of the demo to review are how multiple objects are once again easily created by transforming a master object, and then inserting the transformed version into the rendering list without destroying the original. Of course, you need to keep track of the copies' positions and orientations, but that's nothing more than a couple of arrays of vectors. Now what I would like to do is describe the various elements of the demo and how they work. I'm going to excerpt code from the various parts of the demo because it's too long to list.

Loading and Initializing the Objects

As I said, there are three different types of objects: tanks, towers, and markers. All of them have what I call a *master object* that holds the original. I have secondary data structures to hold the unique information of the copies (which in this case is nothing more that position, because nothing is really moving). Here's the code that loads in and sets up the objects. Notice the scaling I performed to each mesh; sometimes it's necessary to tweak things like this.

```
// load the master tank object
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_tank, "tank2.plg",&vscale, &vpos, &vrot);

// load player object for 3rd person view
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_player, "tank3.plg",&vscale, &vpos, &vrot);
```

Playrix Ex. 1023, page 759

10 0672318350 CH07 05•02•2003 2:42 PM Page 735


```
// load the master tower object
VECTOR4D_INITXYZ(&vscale,1.0, 2.0, 1.0);
Load_OBJECT4DV1_PLG(&obj_tower, "tower1.plg",&vscale, &vpos, &vrot);

// load the master ground marker
VECTOR4D_INITXYZ(&vscale,3.0,3.0,3.0);
Load_OBJECT4DV1_PLG(&obj_marker, "marker1.plg",&vscale, &vpos, &vrot);

// position the tanks
for (index = 0; index < NUM_TANKS; index++)
  {
  // randomly position the tanks
  tanks[index].x = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  tanks[index].y = 0; // obj_tank.max_radius;
  tanks[index].z = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  tanks[index].w = RAND_RANGE(0,360);
  } // end for


// position the towers
for (index = 0; index < NUM_TOWERS; index++)
  {
  // randomly position the tower
  towers[index].x = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  towers[index].y = 0; // obj_tower.max_radius;
  towers[index].z = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  } // end for
```


The Camera System

The camera is fairly simple. It's a Euler model that does nothing more than move around the game field, along with rotating parallel to the y axis. I also could have changed the angle relative to the x axis or the elevation, but the horizon is artificial and I would have had to compute where the artificial horizon rectangles would change when the elevation in 3D changed—too much of a pain! Anyway, here's the code that moves the camera around. Pay attention to the variable `turning`; it's used for an effect.

```
// allow user to move camera

// turbo
if (keyboard_state[DIK_SPACE])
  tank_speed = 5*TANK_SPEED;
```

Playrix Ex. 1023, page 760

10 0672318350 CH07 05•02•2003 2:42 PM Page 736


```
else
  tank_speed = TANK_SPEED;

// forward/backward
if (keyboard_state[DIK_UP])
  {
  // move forward
  cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);
  cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
  } // end if

if (keyboard_state[DIK_DOWN])
  {
  // move backward
  cam.pos.x -= tank_speed*Fast_Sin(cam.dir.y);
  cam.pos.z -= tank_speed*Fast_Cos(cam.dir.y);
  } // end if

// rotate
if (keyboard_state[DIK_RIGHT])
  {
  cam.dir.y+=3;

  // add a little turn to object
  if ((turning+=2) > 15)
   turning=15;

  } // end if

if (keyboard_state[DIK_LEFT])
  {
  cam.dir.y-=3;

  // add a little turn to object
  if ((turning-=2) < -15)
   turning=-15;

  } // end if
else // center heading again
  {
  if (turning > 0)
    turning-=1;
  else
```

Playrix Ex. 1023, page 761

10 0672318350 CH07 05•02•2003 2:42 PM Page 737


```
  if (turning < 0)
    turning+=1;

  } // end else

// generate camera matrix
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

That's basically the entire camera code. It's pretty straightforward. Depending on the keys being pressed, the camera position is moved along a vector based on its current heading or y-axis angle, and remember we are in the x-z, so that's why you see this code to move the camera:

```
cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);
cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
```

Additionally, rotation of the camera is nothing more than updating the `cam.dir.y` variable because it's used by the camera construction call at the end of the code listing. Now the interesting thing is the variable `turning` and how it's used. If you play with the demo, you'll see a tank copy in front of the view screen. This of course represents you, but when you turn right or left, the tank model slowly turns and then turns back to straight when you let off the arrow keys. This is achieved by using the variable `turning` to track the state of the turn in a sense. When the player turns, I let the angle accumulate in the direction of the turn up to some value (15 degrees, in this case) and then it tops out. When the player stops turning, the tank centers itself again. This is a very cool effect used in many racing-type games with a third-person perspective and it takes only a few lines of code. Of course, in a real physics-based simulation, the physics model would do this, but for our purposes, it works fine at this point.

Generating the Game Arena

The game arena is composed of a number of static objects. These objects are copied from master objects and placed at static positions in the game arena. The code for each object basically performs a simple repositioning of the master object and then transforms the master and inserts it into the master polygon rendering list (a cull operation is attempted). The beauty of the system is that the originals are not destroyed. Because only the transformed vertices are used, the original models stay intact. I have two copies of vertices in the object for just such occasions. Here's the code that for example inserts all the tower models into the world:

```
// insert the towers in the world
for (index = 0; index < NUM_TOWERS; index++)
  {
  // reset the object (this only matters for backface and object removal)
  Reset_OBJECT4DV1(&obj_tower);
```

Playrix Ex. 1023, page 762

10 0672318350 CH07 05•02•2003 2:42 PM Page 738


```
  // set position of tower
  obj_tower.world_pos.x = towers[index].x;
  obj_tower.world_pos.y = towers[index].y;
  obj_tower.world_pos.z = towers[index].z;

  // attempt to cull object
  if (!Cull_OBJECT4DV1(&obj_tower, &cam, CULL_OBJECT_XYZ_PLANES))
    {
    // if we get here then the object is visible at this world position
    // so we can insert it into the rendering list
    // perform local/model to world transform
    Model_To_World_OBJECT4DV1(&obj_tower);

    // insert the object into render list
    Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj_tower);
    } // end if

  } // end for
```

Almost too easy, huh? Lastly, let's look at the player insertion model.

Rendering the Player's Tank Model

The engine we have so far isn't very advanced. Although we have a camera, it's not like we can attach the camera to an object and have it tag along. We must do this manually. Alas, in this demo, the position of the camera is known, so what I do is based on the heading of the camera. I draw a vector and then move the tank model out ahead a good distance (400 units approximately) and drop it down from the camera height a little. Then I rotate the tank model (which is a separate model from the one used by the object placer) based on the current viewing angle, transform the tank based on this information, place it in world coordinates and presto—we have a tank that looks like it's us!

The code that does all this magic is rather short:

```
// insert the player into the world
// reset the object (this only matters for backface and object removal)
Reset_OBJECT4DV1(&obj_player);

// set position of tank
obj_player.world_pos.x = cam.pos.x+400*Fast_Sin(cam.dir.y);
obj_player.world_pos.y = cam.pos.y-50;
obj_player.world_pos.z = cam.pos.z+400*Fast_Cos(cam.dir.y);

// generate rotation matrix around y axis
Build_XYZ_Rotation_MATRIX4X4(0, cam.dir.y+turning, 0, &mrot);
```

Playrix Ex. 1023, page 763

10 0672318350 CH07 05•02•2003 2:42 PM Page 739


```
// rotate the local coords of the object
Transform_OBJECT4DV1(&obj_player, &mrot, TRANSFORM_LOCAL_TO_TRANS,1);

// perform world transform
Model_To_World_OBJECT4DV1(&obj_player, TRANSFORM_TRANS_ONLY);

// insert the object into render list
Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj_player);
```

Notice the bolded code. That's where I've added in the slight turning angle along with the current heading to create the effect of a turn.

NOTE

Although I've been using 16-bit graphics for most of the demos, the sad truth is that when we do more serious shaded and textured graphics, it's going to be very hard to keep a decent fill rate with software rasterization and 16-bit graphics. So, we'll do a lot of work with 8-bit modes to get more speed and to see the types of optimizations that can be used with paletted color modes.

Summary

Once again, I think I have worn the fingerprints off my fingers, but it's worth it, baby! There's so much to learn. Sure, you can learn just what you need to make a 3D engine, but I like knowing all the details and all the different ways to do things—I hope you agree. Anyway, we covered a lot of ground in this chapter, such as 3D data structures, loading objects off disk, the local-to-world transform, the world-to-camera transform, the camera-to-perspective transform, the screen transform, object culling, back-face removal, when to use 4D coordinates, why matrices aren't always the best way to perform transforms, rendering objects in wireframe, and finally the UVN camera model and a whole lot of demos! Now, it's time to make our 3D object a little more realistic with shading and lighting. Also, we need to handle 3D clipping because we can't avoid it too much longer. And we definitely need a better model loader!

Playrix Ex. 1023, page 764