

Design and Implementation of Distributed Game Engine

- A Cognitive Simulation Approach

by

Xizhi Li

Shenzhen, P.R. China, 2006

©Xizhi Li, 2006

All rights reserved

Preface

This book is about the design and implementation of a distributed 3D computer game engine, called ParaEngine. Game engines encompass the richest and most advanced technologies in computer science, such as 3D visualization, physics simulation, scripting, networking and AI etc. The composition of these technologies is diverse and constantly evolving. Despite of the relatively large number of game engine implementations, there are no two of them that are alike. This book allows readers to see deep into a unique and commercial quality game engine. Not only does it cover the common and today's popular architectures adopted by many game engines, but it also shows a proven valid framework of a complete game engine implementation.

Many general game development books cover game engine implementation to some extent, but not specific enough to cover details in real world situations, i.e. given a fundamental algorithm or method, how to design efficient software architecture to handle gigabytes of game data at real time. On the other hand, many specialized books cover greater details of a certain aspect of game engine, but novices still need to make risky choices as for which method and how to apply it in their own game engine implementations.

Our book sacrifices some of the completeness on the narration of various game technologies, but saves the space for extra explanation to allow readers to grasp the big idea of each one, examine and evaluate a concrete implementation which have been proven valid in our own game engine. To complement it, most chapters of this book come with a discussion and comparison of various other algorithms not adopted in our implementation, yet can be very useful in different situations. Also, most chapters end with an outlook, which might be a starting point for the next generation engine programmers.

Who Needs This Book

Anyone who wants to understand computer game engine can read this book, because each chapter begins with a carefully written narrative section. It gives the background and methodologies of the chapter subject in plain language and goes into as much depth as possible without much math and source code. **Programmers** will benefit most from this book, because this is one of the few books that cover commercial-quality game engine to the code level. **Researchers and students** who want to apply their innovative ideas in the game domain will find the right spot by reading the outlook section at the end of most chapters. We also hope this book could be used as an educational book teaching game technologies. The author started his first game engine on a university game course without a good book of teaching how. Chapter 2 of the book is thus designed to teach **zealous students** to build their own game engine from scratch in a few days and to extend its functionalities in a spiral development path. However, a complete game engine usually took years and great patience to build.

Assumptions for Programmers

The game engine implemented in this book uses DirectX9.0c and C++. The book does not cover DirectX and 3D math fundamentals, because these things are well documented in DirectX SDK and many places on the Internet. Neither does it require novice engine programmers to know DirectX 3D API and 3D math very well when they read this book. It is

only necessary for them to learn such things when they are doing the actual coding. In fact, even if you think you know the math and DirectX API for a certain program module, you should always hesitate on whether you reinvent the code yourself or doing some research first as directed in this book. Using commercial third-party modules usually saves time and improves quality; and sometimes, even a solution based on open source project is better than writing your own ones. What we are trying to imply to you is that the cost of writing everything from scratch is really high and that some stable game engine modules may be optimized using assembly language six months before their release.

Writing Style

The text in this book is written as concise as possible, which give readers the holistic idea of a computer game engine. The code and logics in this book, however, try to give programmers the concrete picture of how a game engine is written. An engine programmer still need to do its own research as directed in the book in order to implement, but he or she should feel confident enough to apply its knowledge.

About This Book

The book is divided into two volumes. Volume I covers mainly the client side technologies of a computer game engine; volume II covers network related technologies and the cognitive simulation framework of ParaEngine realized through a special language system. Volume I is fit for the design and implementation of a general game engine. Here is a summary of what we covered in volume I of the book.

Chapter 1. Game Engine Introduction

After a short introduction to game engine, it dissects game engine architecture, explains the functions of the different modules of a computer game engine. At the end, it will propose ParaEngine architecture, which will be examined in greater details by chapters 4-19.

Chapter 2. Spiral Development Path

It gives a possible game engine development path. It begins at the spiral center by building up the skeleton of a basic game engine, and then gives the sequence of adding more functionalities and modules to the primitive game engine. This chapter is very useful for people building their own game engine from scratch. In the shortest time possible, it guides you to establish a serious game engine development platform for later extension.

Chapter 3. Files and File formats

To programmers, a software system can be quickly demystified by examine its protocols and file formats. Even for programmers without former game programming experience, it is quite helpful to glance over all file formats used by a game engine. We not only covers the file formats, but also tell you the average file size, average number of each file types processed by a moderate 3D game, and file distributions. Novice programmers and designers can prepare their minds and start reflecting on how large volumes of game data can be managed efficiently at real time. This is a special and independent chapter that we designed to get you very close to a complete computer game engine in a bottom up manner before you even know their implementations.

Chapter 4 – 19. <Content chapters>

Each of the content chapters writes about a specific module of ParaEngine in details and is fairly independent of each other. Each chapter is approximately comprised of the following sections.

1. Foundation: It gives, in plain language, the general idea of the module, basic principles, algorithms and possible solutions, etc. This section usually has sub-sections for sub-systems of the module.
2. Architecture: It only gives the one architecture which is used in ParaEngine, yet in more details.
3. Code and Performance Analysis: It details the architecture section by code or code templates. It also points out the performance critical spots in the code and how to implement them efficiently.
4. Outlooks: It discusses the state of the art of the technology, how newly developed technologies might affect the current implementation of the system and the outlook. It also directs researchers and practitioners to useful resources on the web.

Appendix

We suggest readers read Appendix B to get a quick feel of what a game engine is, before diving into this book.

Using Code in This Book

In general, you can use the code in this book in your programs and documentation. You do not need to contact us for permission unless you are reproducing a significant portion of the code.

We appreciate attribution. An attribution usually includes the author, title, publisher and ISBN. For example: “Xizhi, Li. Design and implementation of distributed game engine. ISBN XXX.”

How to Contact Us

Mail address	Room A811, Shenzhen Tech-Innovation International, Shenzhen,518057, P. R. China
Author's Email	lixizhi@yeah.net
Author's website	www.lixizhi.net

To English Version Readers

When you are reading the English version of this book, you may find that it is not written by a native English speaker. Well, I do hope readers can focus on the content. Yet I apologize for any inconvenience of reading, due to the language. I am the developer of ParaEngine. At the time of writing, I am also the project manager of our game studio. I wrote this book mostly in my spare time, with great patience, using the English language, which is foreign to me. But I find that English is also the only language for this book to be finished in time, because all my former programming notes and reference materials of ParaEngine are in English.

Acknowledgements

First of all, I would like to thank my family for their faith in me. Although father and I both work over twelve hours a day, seven days a week; we can still find plenty of time talking together every day. Father has given me continuous supply of inspirations. And it is my great mother who has given both father and me wisdom and strength.

I would also like to thank all formal and informal team members of ParaEngine Dev Studio. They are Li Lixuan, Wang Tian, Liu Weili, Li Yu, Liu He, Zhang Yu and dozens of friends from college. They have contributed time and ideas to ParaEngine as well as to its first few application demos. Many thanks to professors and crews at Shenzhen Tech-Innovation International (STI). They have provided me the workplace of our studio and financial support since my graduation.

During the past years, I have introduced my ideas and works to quite a few University professors as well as experienced game developers. They have given me valuable opinions. They are Prof. Shu Wenhao, Prof. He Qinming (my advisor at Zhejiang University), Mr. Xu Zhendong. (*TODO: add more reviewers to this list*)

Finally, I would like to thank my tutor Lu Yang, who had given me great teachings on computer programming for ten consecutive years since 1989.

Table of Contents

Who Needs This Book.....	ii
Assumptions for Programmers	ii
Writing Style	iii
About This Book	iii
Using Code in This Book	iv
How to Contact Us	iv
To English Version Readers	iv
Acknowledgements	v
Chapter 1 Game Engine Introduction.....	1
1.1 Game Engine Technology and Related Research.....	1
1.1.1 A Glance at Computer Game Engine	2
1.1.2 Games as a Driving Force	3
1.2 Introduction to Distributed Computer Game Engine.....	5
1.3 ParaEngine Overview	5
1.3.1 Foundation Modules	5
1.3.2 The Game Loop.....	7
1.3.3 Summary	8
Chapter 2 Spiral Development Path	10
2.1 A Basic Skeleton of Computer Game Engine	10
2.2 Spiral Development Path	11
2.3 Building Game Engine Skeleton	12
2.3.1 Preparation.....	12
2.3.2 Writing the Game Loop.....	13
2.3.3 Time Controller	16
2.3.4 Log System.....	17
2.3.5 Memory Leak Detection.....	18
2.3.6 Using Mini-Dump	19
2.3.7 File Manager.....	19
2.3.8 Asset Manager	20
2.3.9 Scene Manager	25
2.3.10 Scripting System.....	26

2.4 Summary and Outlook.....	28
Chapter 3 Files and File formats	29
3.1 File Format from Game Requirement	29
3.2 ParaEngine File Formats Overview.....	30
3.3 ParaEngine File Formats	32
3.3.1 World Config File.....	32
3.3.2 Terrain Config File	33
3.3.3 Terrain Elevation File.....	34
3.3.4 Terrain Mask File	34
3.3.5 Terrain On Load Script.....	35
3.3.6 Texture File	36
3.3.7 Mesh and Animation File	36
3.3.8 Archive File	41
3.3.9 NPL Script File.....	42
3.4 Summary and Outlook.....	43
Chapter 4 Scene Management	44
4.1 Foundation.....	44
4.1.1 Hierarchical Scene Management.....	44
4.1.2 Scene Graph.....	47
4.1.3 Using Scene Manager	47
4.1.4 Dynamic Scene Loading and Unloading	48
4.2 Scene Manager Architecture	50
4.2.1 The Scene Root Object.....	50
4.2.2 Relative Position VS Absolute Position	51
4.2.3 Quad Tree Terrain Tiles	51
4.3 Code and Performance Discussion	52
4.3.1 Quad-tree Balance Discussion.....	53
4.4 Summary and Outlook.....	53
Chapter 5 Rendering Pipeline.....	55
5.1 Foundation.....	55
5.1.1 Render Pipeline Basics	55
5.1.2 Object Level Clipping with Bounding Volumes	57
5.1.3 Object Visible Distance	57

5.1.4 Coordinate System for Render Pipeline	58
5.1.5 Hardware Occlusion Testing	59
5.2 Building the Render Pipeline.....	60
5.3 Shader Program Management	62
5.3.1 Effect File Manager.....	62
5.3.2 Wrapping the Render API	64
5.4 Code and Performance	65
5.4.1 Traversing Quad Tree.....	65
5.4.2 Render Coordinate System Transformation	66
5.4.3 Hardware Occlusion Testing	67
5.4.4 Effect Management	67
5.4.5 Performance Discussion	70
5.5 Summary and outlook.....	71
Chapter 6 Special Scene Objects	73
6.1 Sky.....	73
6.2 Sun and Sun Light	73
6.3 Particle System	74
6.3.1 Defining a Particle System	74
6.3.2 Animating Particles	76
6.3.3 Drawing Particles	79
6.4 Weather System.....	81
Chapter 7 Picking	82
7.1 Foundations	82
7.1.1 Background.....	82
7.1.2 Picking Mathematics	83
7.2 Architecture and Code	87
7.2.1 Ray Picking in the Scene	87
7.2.2 Collision Detection Math Code	89
7.3 Summary and Outlook.....	91
Chapter 8 Simulating the Game World	92
8.1 Foundation.....	92
8.1.1 Rigid Body Dynamics	92
8.1.2 Sensor Ray based Physics	95

8.1.3 Motion Blending.....	100
8.1.4 Autonomous Animation	101
8.2 Architecture and Code.....	102
8.2.1 Integrating the Physics Engine	103
8.2.2 Environment Simulator	104
8.2.3 Sensor Ray Structure	107
8.2.4 Ray based Character Simulation.....	108
8.3 Summary and Outlook.....	114
Chapter 9 Terrain.....	116
9.1 Foundation.....	116
9.1.1 Understanding Terrain Data	116
9.1.2 Level of Detail Terrain Algorithm.....	118
9.1.3 The Hybrid Terrain Algorithm	121
9.1.4 Painting the Terrain	126
9.1.5 Miscellaneous Functions	127
9.1.6 Latticed Terrain	128
9.2 Architecture and Code	130
9.2.1 Tessellation Code	130
9.2.2 Building Triangle Buffer	133
9.3 Summary and Outlook.....	135
Chapter 10 Ocean	136
10.1 Foundation.....	136
10.1.1 Simulation of Water Surface	136
10.1.2 Rendering of Water Surface	140
10.2 Architecture and Code	145
10.2.1 Ocean Rendering Pipeline	145
10.2.2 Shader code for Reflection and Refraction mapping.....	146
10.3 Summary and Outlook.....	150
10.3.1 Shorelines	150
Chapter 11 Special Effects	152
11.1 Shadows.....	152
11.1.1 Static Shadow	152
11.1.2 Dynamic Shadow.....	152

11.1.3 Code.....	159
11.2 Light Mapping.....	163
11.2.1 Light Mapping Basics.....	163
11.2.2 Consider Baked Texture First.....	164
11.2.3 When to Use Light Map	164
11.2.4 Cost of Using Light Maps	165
11.2.5 Retrieving Light Map Texture Coordinates from 3dsmax	165
11.3 Cubic Environment Mapping	168
11.4 Billboarding.....	169
11.5 Reflective Surfaces.....	170
11.6 Local Lights.....	171
11.7 Full Screen Glow Effect	172
11.7.1 Code.....	175
11.7.2 Discussion.....	179
11.8 Fog Effect	180
11.8.1 Fog and Sky	180
11.8.2 Fog with Global Sun Lighting	183
11.8.3 Fog underwater.....	184
11.8.4 Fog with Large Mesh Objects	184
11.9 Conclusion.....	184
Chapter 12 Character and Animation	186
12.1 Foundation.....	186
12.1.1 Background of Motion Synthesis	186
12.1.2 Introduction to Skeletal Animation	187
12.1.3 Motion Blending.....	190
12.1.4 Character State Manager	193
12.1.5 Attachment and Custom Appearance	195
12.2 Architecture	196
12.2.1 Model File	196
12.2.2 Animation Instance.....	196
12.2.3 State Manager.....	197
12.3 Code and Performance Discussion	197
12.3.1 Spherical Linear Interpolation of Quaternion.....	197

12.3.2 Retrieving Animation Keys.....	198
12.3.3 Animation Instance Interface.....	201
12.3.4 Character State Manager	202
12.4 Summary and Outlook.....	204
Chapter 13 Writing Model Exporters	206
13.1 Foundation.....	206
13.1.1 Terminologies.....	206
13.1.2 Exporting Mesh	211
13.1.3 Exporting Animation Data	211
13.1.4 Importing Animation	213
13.2 Architecture	215
13.2.1 Exporter Interface	216
13.2.2 Scene Data Extraction	217
13.2.3 Serialization of Model Data.....	217
13.3 Code and Performance Discussion	218
13.3.1 Sampling Bone Animation	218
13.3.2 BVH Serialization Code	220
13.3.3 Performance Discussion	223
13.4 Summary and Outlook.....	223
Chapter 14 AI in Game World	225
14.1 Foundation.....	225
14.1.1 Reactive Character.....	225
14.1.2 Movie Character	226
14.1.3 Perceptive Character.....	226
14.1.4 Character Simulation and Sensor Events.....	226
14.1.5 AI Controllers.....	232
14.1.6 Frame Rate of AI Logics	234
14.1.7 Summary of AI	236
14.2 Architecture	236
14.2.1 Game Object Base Class	236
14.2.2 AI Controller	237
14.2.3 Other AI Modules.....	239
14.3 Code and Performance Discussion	239

14.3.1 Follow Controller	239
14.3.2 Sequence Controller	241
14.4 Summary and Outlook.....	247
Chapter 15 Navigating in 3D world	249
15.1 Camera Control	249
15.1.1 Background.....	249
15.1.2 Occlusion Constraint	249
15.2 Generic Path Finding Algorithms.....	253
15.2.1 Object Level Path Finding.....	253
15.2.2 Path-finding and Physics	255
15.2.3 Code.....	255
15.3 Summary and Outlook.....	258
Chapter 16 NPL Scripting System	259
16.1 Foundation.....	259
16.1.1 Using LuaBind.....	259
16.1.2 Exporting C++ Attributes to Script	261
16.1.3 Runtime State Management.....	263
16.1.4 Using Namespace	264
16.1.5 Using Directory and Domain Name	265
16.1.6 Script File Activation Mechanism.....	266
16.2 Architecture and Code	267
16.2.1 Attribute Field	267
16.2.2 Attribute Class	269
16.2.3 Attribute Field Interface	270
16.2.4 Attribute Class Script Wrapper.....	272
16.2.5 Script Examples.....	273
16.3 Summary and Outlook.....	273
16.3.1 NPL Feature Overview.....	273
16.3.2 Summary and Outlook of NPL.....	274
Chapter 17 File System	275
17.1 Foundation.....	275
17.2 Architecture	276
17.3 Code.....	278

17.4 Summary and Outlook.....	280
Chapter 18 Frame Rate Control.....	281
18.1 Foundation.....	281
18.1.1 Overview	281
18.1.2 Decoupling Graphics and Computation	282
18.1.3 I/O.....	282
18.1.4 Frame Rate and Level of Detail.....	283
18.1.5 Network servers.....	283
18.1.6 Physics Engine.....	284
18.2 Architecture	284
18.2.1 Definition of Frame Rate and Problem Formulation.....	285
18.2.2 Integrating Frame Rate Control to the Game Engine	286
18.3 Evaluation.....	287
18.3.1 Frame Rate Control in Video Capturing.....	288
18.3.2 Coordinating Character Animations	288
18.4 Summary and Outlook.....	289
Chapter 19 Summary and Outlook of Game Engine	290
19.1 Age of Middleware.....	290
19.2 World Editing	290
19.2.1 In-game Editing	290
19.2.2 Customizing over Professional 3D Authoring Tools.....	290
19.2.3 Writing Special Tools for Game.....	291
19.2.4 Outlook	291
19.3 Next Generation Game Engine.....	291
Appendix A : Code Optimization.....	292
Appendix B : Da Vinci World	294
B.1 Game Proposal and Design.....	294
B.2 Prepare Your Work Environment	295
B.3 Getting Used to ParaEngine SDK.....	295
B.4 Creating a New World	297
B.5 Making Game Scene	299
B.6 Do It Yourself: Advanced Terrain	301
B.7 Do It Yourself: Static Models.....	302

B.8 Do It Yourself: Characters	305
B.8.1 Special Rules.....	306
B.9 Adding Artificial Intelligence	308
B.9.1 Do it yourself: Artificial Intelligence.....	308
B.9.2 Do it yourself: Using AI Controllers	311
B.10 Do It Yourself: User Interface	314
B.10.1 Game Loop File	314
B.10.2 UI Controls	314
B.10.3 IDE libraries.....	315
B.11 Release	315
B.12 Conclusion	315
Appendix C About ParaEngine	316
C.1 Introduction.....	316
C.2 ParaEngine Specifications	316
C.3 About ParaEngine Dev Studio.....	319
Appendix D Figures in the Book.....	320

Chapter 1

Game Engine Introduction

Game engine is an all encompassing subject in computer science. In the narrow sense, a game engine is a fairly general game development platform which can be used to build many games. A modern game engine can be regarded as a virtual reality framework running not only on a standalone computer, but also on a computer network. It interacts with human users with multimedia input and output, simulates the networked virtual environment with laws similar to the real world, and animates characters which exhibit approaching human-level intelligence. Beneath the interface, a game engine does almost everything that a computer is capable to do and it must do it in the fastest way possible. Game technology represents the richest virtual reality implementation that present day hardware could afford at real time.

In recent years, game engine related technology has drawn increasing academic attention from a wide range of research areas. People come to realize that game engine may naturally evolve in to the most widely used virtual reality platform in the future. The topic of the book is on the design and implementation of a distributed computer game engine called ParaEngine. ParaEngine is a commercial quality game engine aiming to bring interactive networked virtual environment to the Internet through game technologies. In ParaEngine, game worlds and logics on individual computers can be linked together like web pages to form an interactive and extensible gaming environment.

In a computer game engine, there are tremendous number of wisdoms and choices in putting all its components into one piece of working software. During the past years, new algorithms and hardware evolutions have made several big impacts on the integration of game technologies in a computer game engine. It is safe to predict that game technology will continue to be a rapidly reforming area in the future.

1.1 Game Engine Technology and Related Research

Since early 1990s, game engine has evolved to become a standard platform for constructing and running 3D virtual game worlds of high complexity and interactivity. In recent years, game engine research has drawn increasing academic attention to it, not only because it is highly demanding by quick industrial forces, but also because it offers mature and extensible platform support for a wide range of researches in computer science and engineering. These includes computer graphics, autonomous animations, web avatars, artificial intelligence, virtual reality (VR) and augmented reality (AR), networked VR, stereo vision, HCI, web 3D technologies, multi-agent framework, distributed computing and simulation, education and military training, robotics, etc.

The term "game engine" arose in the mid-1990s, especially in connection with 3D games such as first-person shooters (FPS). Such was the popularity of id Software's Doom and Quake games that rather than work from scratch, other developers licensed the core portions of the software and designed their own graphics, characters, weapons and levels—the "game content" or "game assets." Later games, such as Doom 3 and Epic's Unreal were designed with this approach in mind, with the engine and content developed separately. The continued refinement of game engines has allowed a strong separation between rendering, scripting, artwork, and level design. Modern game engines are some of the most complex applications

written, frequently featuring dozens of finely tuned systems interacting to ensure a finely controlled user experience.

1.1.1 A Glance at Computer Game Engine

Game engine is a very broad subject. The part that deals with graphic storage and displaying is often called 3D engine or rendering engine. Rendering engine is usually the core component in a present day game engine; but a game engine usually includes support for many more aspects of a game's architecture than just the 3D rendering. Some part of game engine may be implemented by "middleware". Middleware developers attempt to "pre-invent the wheel" by developing robust software suites which include many elements a game developer may need to build a game. Most middleware solutions provide facilities that ease development, such as graphics, sound, physics, biped animation, networking and AI functions. E.g. physics engines such as Novodex¹ and Havok² are commonly used by top commercial game engines to construct physically convincing games. In the viewpoint of a game development corporation, integrating middleware in their game engine is usually called outsourcing³.

The continued refinement of game engines has allowed a strong separation between rendering, scripting, artwork, and level design. It is now common (as of 2006), for example, for a typical game development team to be composed of artists and programmers in an 80/20 ratio. Hence, game engine should also include a complete suite of virtual world constructing tools, which are used mostly by artists. For example, 3D models, key framed animations, physical specification of game objects, etc are usually made by professional tools such as 3DsMax and Maya, and then exported to engine digestible file format by exporters provided by the game engine; cinematic, game levels, etc may also be made in a similar way, whereas some game engine provides more specialized tools, such as visual script editor, audio editor, model editor, level editor, et., which allow artists and level designers to compose game scenes on the fly (i.e. graphics attained at design time appear exactly as in the final game). For example, Quake series level editing tools are popular among hobbyists and also used optionally by some open source game engines⁴.

Table 1.1 shows a quick a quick jot-down list of game engine technologies.

Table 1.1 A quick jot-down list of game engine technologies

Category	Items
Graphics	scene hierarchy, skinning, shadow / lighting model, particle systems, shader model / material system / associated tools, vertex and skeletal animation, alpha/textture sorting, terrain, clipping/culling/occlusion, frame buffer post-processing effects, split-screen support, mirrors/reflection,

¹ Novodex physics engine, <http://www.ageia.com/novodex.html>

² Havok solutions, <http://www.havok.com>

³ Outsourcing Reality: Integrating a Commercial Physics Engine, Game Developer Magazine, 2002.

⁴ 3D Engines Database, <http://www.devmaster.net/engines/>

	procedural geometry, 2D GUI (buttons, text rendering/font issues, scrolling windows, etc), level of detail, projected textures, ...
Resource management	Dynamic loading, resource lifetimes/garbage collection, streaming resource scheduling, rendering device change management, file access (virtual files)...
sound/music	3D, 2D, looping, looping sub range, effects, ...
in-game UI	also related to graphics and scripting.
I/O	key remapping, force feedback (haptic devices)
time management	frame rate control: time synchronizations with various engine modules
Scripting	Lua/python/?, saving and loading game state, security, performance, profiling, compiler and debugging, in-game cinematic, ...
Tools	level editor, terrain editor, particle system editor, model/animation viewers trigger system tool, cinematic tool, MAX plug-ins – exporter, ...
Console	In game: debugging, in-game editor, recording/playback: frame-based, time-based compatibility: cross-platform compatibility, graphic device support
Networking	distributed client/server, single client/server, peer-to-peer, security/hacking issues, package misordering, time synchronization, delay, bandwidth usage, error handling ...
Physics	stable physics integration, frame rate control, outsourcing physics engine?, collision detection (continuous or discrete) , collision response (approximation or impulse-based), integration with key framed animation, line of sight/ray queries, ...
Animation	inverse kinematics, key framed animation, motion warping/blending,
AI	fuzzy logic, machine learning, state machines, path-finding, tools (scripting)...
General	memory management, exception handling, localization, enhancing concurrency/multi-threading, ...

The four major modules in game engine are 3D rendering (graphics), scripting, physics simulation and networking, as shown in bold text in the table. This framework design as well as individual component implementations decides the general type of games that could be composed by it.

1.1.2 Games as a Driving Force

The study of a game engine framework is now gaining increasing popularity among the academic community for at least the following reasons.

- The knowledge of a computer game engine will greatly help the design and implementation of a multimedia application in the future, since user interface application will become more and more playable and user-friendly like computer games. Moreover, a game engine framework has many good design patterns and common libraries which developers can reuse.
- It is an area of research where software and hardware developers work closely together. In other words, it has many hardware peripherals which it depends on, such as the network server facilities, the haptic devices, the stereo glasses, the graphics/physics acceleration cards, and other parallel architecture that increases the computing power of a modern personal computer. Most hardware improvements made could immediately be put to use in a computer game engine.
- It is industrial demanding. As people have more time and less space, they turn to the virtual world for resorts and accomplishment. The biggest benefit that computers bring to mankind is likely to be virtual reality. More and more real world services will be built in to future virtual reality framework. Hence, the mastery of game engine tools and framework will be as important in the future as building web pages today.
- Games are the driving force to push computer technologies and to bring the Internet from 2D to 3D. Unlike other industrials, games and game technologies are exposed to the largest number of users and it is especially favored by young people. Both playing and developing games are community based activities that last for a long time. All these attributes make game engine a promising research area to push computer technologies in a variety of fields.
- Game engine is a highly interdisciplinary study. It evaluates the performances and effects of several candidate approaches in a certain research area, and designs the combinations of the best choices from many areas of study, which in turn will help refining the approaches in separate disciplines.
- Game technologies are being applied to a number of other fields, such as remote education, digital learning, augmented reality guidance system, military training, medical treatment, vehicle training, scientific simulation and monitoring, and animations in movies, etc.
- Games are also a driving force for a variety of other serious research areas, such as parallel and concurrent computing and programming, artificial neural networks, autonomous character animations, human computer interface, human cognitions, robotics, etc.

Finally, “Computer Science alone was not sufficient to build our future modeling and simulation systems.” (Zyda). Game content developers must acquire other cross-disciplinary skills to build well-qualified virtual environment. Game development is both technology and art intensive. The 2004 game report⁵ provides timely analysis and actionable insights into emerging game technologies and their potential impacts on existing and new technical education curricula.

⁵ Jim Brodie Brazell, Digital Games: A Technology Forecast, Texas State Technical College, 2004.

1.2 Introduction to Distributed Computer Game Engine

“Distributed Game Engine is a game engine framework which allows virtual world content and game world logics to be distributed on a large scale and extensible computer network, such as the Internet.”

ParaEngine is a distributed computer game engine. But it does not complicate the issue of introducing game engine technologies, because it shares all the common features of a general computer game engine and have more flexible and extensible architecture design.

The game genre that distributed game engine advocates is open games, of which the gaming environment extends as far as the network could reach. For example, initially a publisher may host a 3D game world on a small cluster of servers on the Internet; as more users are playing its game, the publisher may frequently increase the size of the game world and the scale of game servers; meantime, players may also help to extend the gaming environment with its own computers. In the future, games will be as open as the Internet; its supporting framework may be called distributed game engine.

Plainly speaking, if we compare web pages to 3D game worlds, hyperlinks and services in web pages to active objects in 3D game worlds, and web browsers and client/server side runtime environments to computer game engines, we will obtain the simplified picture of distributed computer games. It is likely that one day the entire Internet might be inside one huge virtual reality game world.

1.3 ParaEngine Overview

This section shows one possible division of game engine functionalities into fairly independent program modules. It also shows the game loop that calls these modules. Programmers who are new to engine programming need not fear about the number of modules presented here, since we will guide you in a spiral development path to build a serious game engine from the very beginning.

1.3.1 Foundation Modules

The core of ParaEngine is comprised of the following modules.

- Asset Manager: It manages all kinds of game resources and device objects used in the game engine, such as textures, geometries, sounds, effects, buffers, etc. As a general rule, the manager provides a unified interface to automatically load an asset from disk file only when they are used and remove those that are not needed.
- Scene Manager: It manages all game objects which comprises the local 3D game world. Usually, all game scene objects are organized in a tree hierarchy (usually called scene graph), with a root node on top called scene root. The rendering pipeline usually builds objects seen in a frame by traversing the scene graph. The following modules are also child objects of the scene manager.
 - Global Terrain Engine: For efficient rendering of an infinitely large terrain surface. It also provides physics query interface, such as getting the height of terrain at a specified location.

- Ocean Manager: For realistic and efficient rendering of water and ocean from under and/or above the water surface. It provides physics query interface.
- Physics World: It is a wrapper of physics engine. All other physical objects except the terrain and ocean are managed in this place. These are usually solid mesh objects and basic geometries for collision detection and response in the virtual world.
- Biped State Manager: It generates smooth and valid animations (actions) for a character according to its physical interaction with the environment, such as jumping, walking, swimming, etc.
- Biped Controllers (or AI modules): Some higher level character behaviors, such as object following, face tracking, action recording and replaying, etc are encapsulated in several biped controllers, which can be dynamically assigned to characters at runtime.
- Camera system: The camera system provides a view into the game world. It can be made to automatically position itself to avoid collision with the physical environment and focused on a scene object, such as the current player.
- GUI Engine: It manages 2D GUI objects such as buttons, text, edit boxes, scrollable windows, etc. It also handles mouse and keyboard events in the GUI.
- Environment Simulator: At each time step, the environment simulator advances the game state by a small interval. It takes input from the user, AI modules, and the scripting system, validates actions issued by mobile game entities, updates their impacts in the local virtual world, computes the new game states according to some predefined laws, and feeds environmental perception data to AI modules and the scripting system, etc.
- Scripting system (or NPL runtime environment as called in ParaEngine): Game contents and logics can be expressed in external files, which are compiled and executed on the fly without modifying the game engine itself. We call this technique scripting. The language we used in ParaEngine's script files is called NPL. The module that parses and executes these script files is called NPL runtime environment. Game content, such as 3D scenes and graphic user interface, and logics such as character AI and user IO, can all be written and controlled entirely from script files. In NPL, network behaviors are also implemented in this module, which makes writing and deploying distributed games easy.
- Game Loop: It drives the flow of the engine, providing a heartbeat for synchronizing object motion and rendering of frames in a steady, consistent manner.
 - Frame rate controller: it is a time management module which supplies other game engine modules with a steady time step.

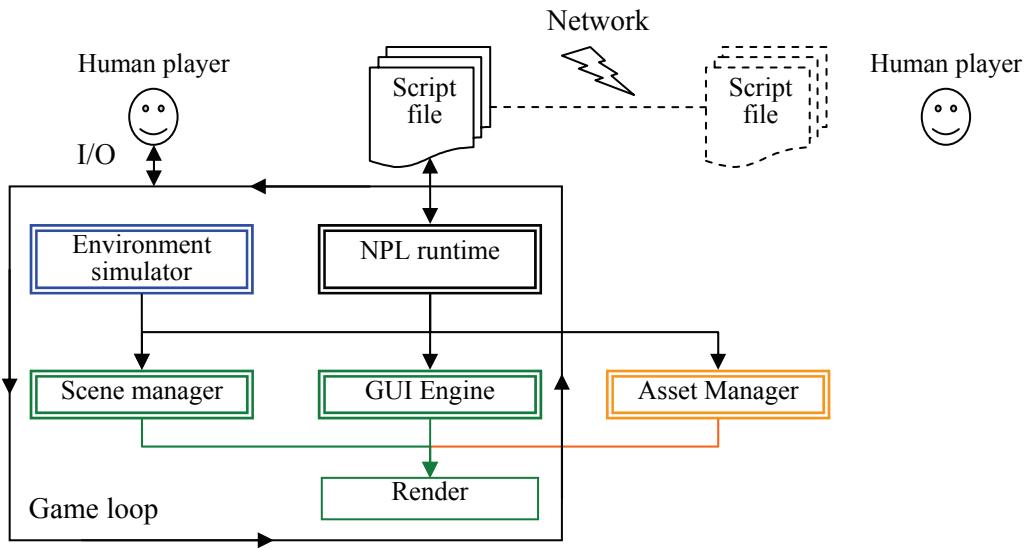


Figure 1.1 Overview of the major engine modules

1.3.2 The Game Loop

When designing the functioning core of an interactive software system, we usually start from its main loop. This has not changed much even with the use of multi-threading, message driven and event based system. The main loop in a computer game engine is called game loop, which drives the flow of the engine, providing a heartbeat for synchronizing object motion and rendering of frames in a steady, consistent manner. A general game loop is given in Table 1.2. It has been extended to include more details of the engine framework.

Notes: A TIMER is a special object that decides how many times that a dependent module should be executed in that loop iteration. For example, one can customize a timer to produce 30 beats per second with equal time steps. We will cover it in more detail in the frame rate control chapter.

Table 1.2 Game loop

<pre>Main game loop callback function { Time management: update and pre calculate all timers used in this frame. Process queued I/O commands (IO_TIMER) { Mouse key commands: ray-picking, 2D UI input Key stroke commands Animate Camera (IO_TIMER): Camera shares the same timer as IO } Environment simulation (SIM_TIMER) { Fetch last simulation result of dynamic objects } }</pre>

Validate simulation result and update scene object parameters, accordingly.

Update simulation data set for the next time step:

Load necessary physics data to the physics engine; unload unused ones.

Calculate kinematic scene objects parameters, such as player controlled character (this usually results from user input or AI strategies.).

Update necessary simulation data affected by kinematic scene objects.

Start simulating for the next time step (this may run in a separate thread than the game loop).

Run AI module (SIM_TIMER) {

Run game scripts (SIM_TIMER): Currently **networking** is handled transparently through the scripting system.

}

}

Render the current frame (RENDER_TIMER) {

Advance local animation (RENDER_TIMER)

Render scene (RENDER_TIMER)

Render 2D UI: windows, buttons ...

}

In-game video capturing (RENDER_TIMER)

}

Besides the game loop, there may be other game threads running. These include: the Windows message handler, NPL runtime environment, network threads and physics engine. In most cases, Windows message handler and NPL runtime are running in the same thread as the game loop.

1.3.3 Summary

This chapter briefly reviews game engine and its applications. It immediately proposes our game engine architecture and explains the functions of each major program module. Details of the game engine architecture will be examined in later chapters.

This book, together with its cited works, is described at a level of detail that allows for researchers and practitioners to design and build the next generation distributed computer game engine. We hope that the concept of distributed game engine can be accepted and more people will continue this concrete work with their updated visions.

The code and architecture presented in this book is based on a formal game engine implementation called ParaEngine. Therefore, they are guaranteed to function well with current computer hardware.

The major contributions of this book (for volume I and II), we hope, are the following:

- A big picture of distributed game engine platform from a practitioner's viewpoint, explained in plain language.
- A concrete introduction to game engine with a proven implementation covering every corner of a formal game engine.
- Accurate redirections to other game programming and research resources on the web.
- Addressing a few rarely discussed problems in distributed game engine design, such as:
 - Simultaneous visualization and simulation of distributed game world content and logic
 - Scripting language runtimes in distributed virtual environment
 - Time management in distributed game worlds
 - Autonomous character animation generation

Chapter 2 Spiral Development Path

This chapter gives a possible game engine development path. It begins at the spiral center by building up the skeleton of a basic game engine, and then gives the sequence of adding more functionalities and modules to the skeleton game engine. This section is useful for people building their own game engine from scratch. In the shortest time possible, it guides you to establish a serious game engine development platform for later extension.

Unlike other software system, during the development of a computer game engine, many modules and functions may need to be rewritten several times, not only because the requirement of a game engine changes fast but also because game technologies are evolving faster. This is one reason why game engine development team, even the most successful ones, is relatively small, with only one or two chief programmers.

2.1 A Basic Skeleton of Computer Game Engine

Figure 2.1 shows a very basic skeleton of computer game engine. Although its only function is displaying mesh object via script interface, it has the most basic modules as in a formal game engine. Starting from the beginning, we need a game loop and a time controller. We need an asset manager to manage textures and meshes. We need a log system for debugging and performance analysis. We will have a unified file access interface for file management. We need a scene manager to store mesh objects in a game scene. We need a camera system to store camera parameters, such as eye position and orientation. We need a scripting system, so that we can design some script interface to allow loading scene mesh objects from script. Finally, we need two dummy modules, called GUI Engine and Environment Simulator, which do nothing in this first implementation.

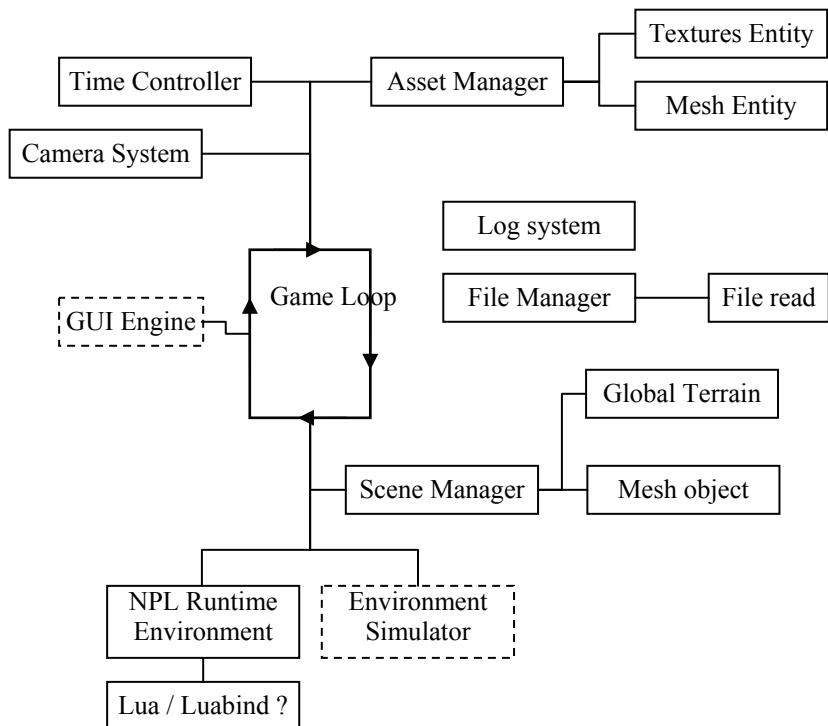


Figure 2.1 Basic Skeleton of a Computer Game Engine

In this chapter, we will guide you to build this skeleton engine. But before that, we will first take a look at a more complete version of the spiral development path, that this book will guide you to complete.

2.2 Spiral Development Path

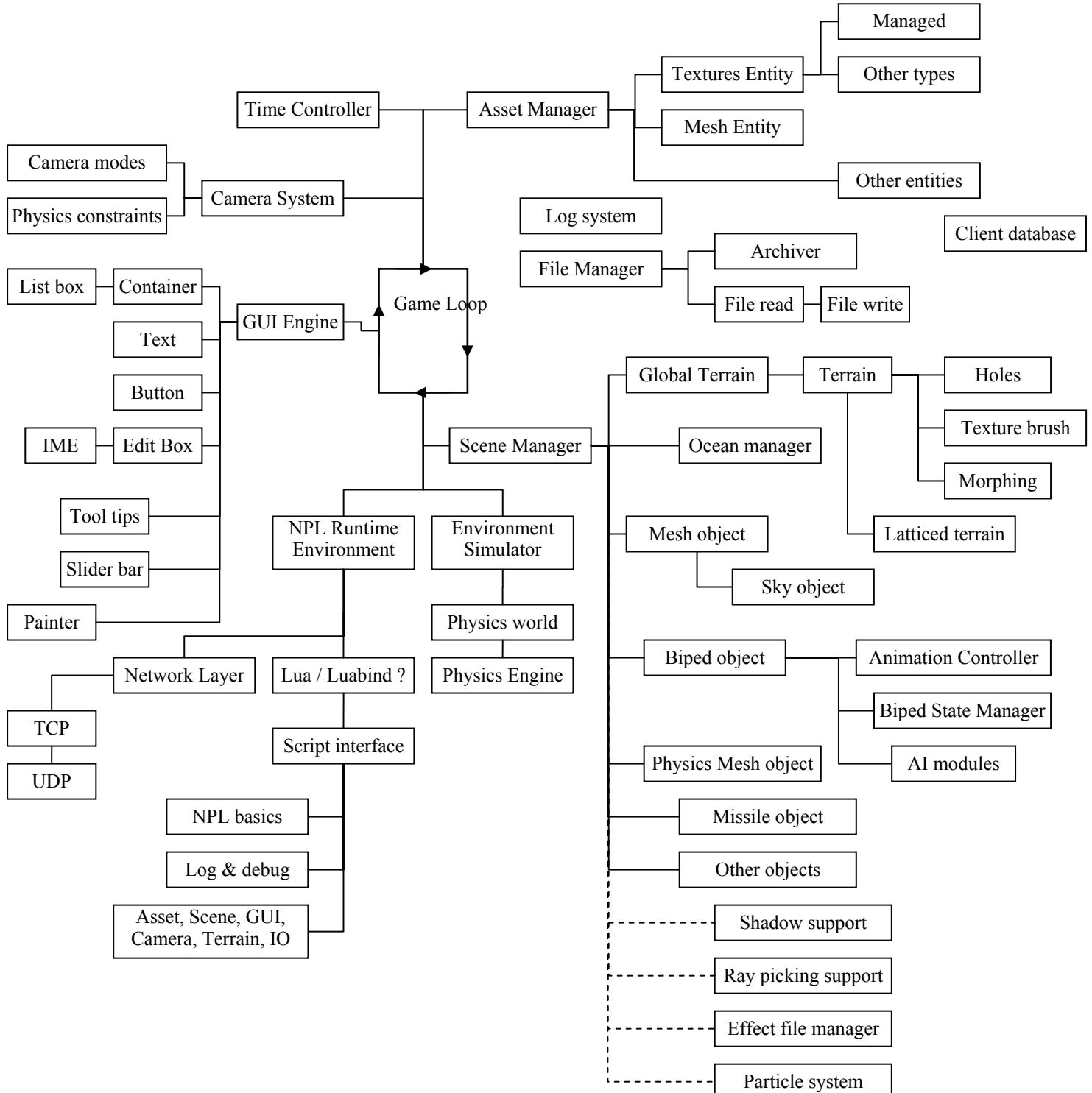


Figure 2.2 Spiral development path

Figure 2.2 is an extension of Figure 2.1. Generally speaking, engine developers can start from the center game loop and then adding new functionalities following the lines in the figure. The longer the horizontal lines, the later the connected module may be developed. For example, after having implemented a basic Scene Manager, one can build the Global Terrain; then a single height field based Terrain; then a Latticed Terrain; and then adding terrain holes, texture brushes and terrain morphing functionalities, etc. The implementation details for the extension parts can be found in the corresponding chapters of this book.

2.3 Building Game Engine Skeleton

This section guides you to build the game engine skeleton in Figure 2.1. We use C++ as the main programming language, so that we can take advantage of object oriented design, C++ template and namespace support. DirectX is assumed as the rendering device API. One can download all source code covered in this section from our website: www.paraengine.com

2.3.1 Preparation

The following things are almost absolutely necessary if you are a reader who wants to start building a modern game engine with C++. These things will not be explained in this book, but worth learning elsewhere.

- C++ Namespace is very useful for a big game engine project, since we may later use third-party libraries which might have name collisions with our own classes or data types. So we advise engine developers to put every class, function or global object that belongs to the game engine to a namespace. Our sample game engine is called ParaEngine, so we put everything in namespace ParaEngine.
- Getting familiar with STL, we will need its implementations for a number of data structures and algorithms. Alternatively, one should at least prepare its own efficient and safe counterparts for STL string, list, vector, map and set, etc.
- Getting familiar with string safety. Game engine will do lots of IO and string manipulations. And one should make sure how to use functions like scanf, printf, string length, etc in a secure and efficient way. More information can be found at Microsoft MSDN on code security.
- Learn to document your source code. Game engine will evolve to a huge project and always subject to changes; code documentation can provide timely updates to function usage for the programmer itself and other users. In ParaEngine, we use *doxygen*⁶ format and only document in the header files and the function bodies.
- DirectX basic programming. One does NOT need to be very familiar with DirectX or OpenGL API to begin his or her first game engine. But at least, one should be able to understand the basic pipeline and math to render a simple 3D triangle anywhere in the window. This is the pre-requisite of this book for programmer readers.

⁶ Doxygen, a documentation system: <http://sourceforge.net/projects/doxygen/>

- Scripting. This is not necessary for every game engine. But since we use an open source language called Lua in our scripting implementation, it becomes necessary for programmers to download and learn something about Lua and LuaBind. Lua is a hidden secret in the game industry for decades and has been used in a number of successful games. We will cover more on it both in this chapter and the chapter for scripting.
- Math library. 3D game engine will do lots of vector, quaternion and matrix math, etc. Many game engines have their own math libraries. DirectX SDK also provides a bunch of inline helper functions (prefixed with D3DX) to perform math functions. Although DirectX's math functions are not of object oriented design, it is general enough to be used together with other math libraries. Hence, most math of ParaEngine is done with DirectX helper functions, except for special conditions. ParaEngine's own math library can be found at the book's website; other good object-oriented math libraries can be found in some open source projects, such as the ODE physics engine⁷.

2.3.2 Writing the Game Loop

Do not use the operating system's timer for your game loop. Game engine will need its own more accurate timing module. Usually, it is good practice to put your game loop in your main application's while (true) {} block and let it be called as often as possible. In a Win32 application, it looks like the following code:

```
class CMyD3DApplication : CParaEngineApp {
    ...
    INT run(){
        While(if app does not quit){
            if(got a message){
                ... Translate and dispatch the message
            }else{
                Render3DEnvironment(); // game loop in this function
            }
        }
    };
    INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT ){
        CMyD3DApplication d3dApp;
        InitCommonControls();
        if( FAILED( d3dApp.Create( hInst ) ) )
            return 0;
        return d3dApp.Run();
    }
}
```

A real game loop takes more code than that, we advise novice programmers to refer to DirectX samples or the code provided in our book's website. In case, one wants to use the same game engine for different applications, we use a special class called CParaEngineApp to enclose all application specific functions. In future, we can optionally build our project as a library file to be used in many game projects, which call the CParaEngineApp' class API in their own game loops. See the code sample below.

```
class CParaEngineApp{
public:
```

⁷ Open Dynamics Engine: www.ode.org/

```

/** this function should be called when the application is created. I.e. the windows HWND is valid.*/
HRESULT OnCreateWindow();

/** Called during device initialization, this code checks the device for some minimum set of capabilities,
and rejects those that don't pass by returning false.*/
HRESULT ConfirmDevice(LPDIRECT3D9,D3DCAPS9*,DWORD,D3DFORMAT,D3DFORMAT);

/** This function should be called only once when the application start, one can initialize game objects
here.*/
HRESULT OneTimeSceneInit(HWND* pHWND);

/** This function will be called immediately after the Direct3D device has been created, which will happen
during application initialization and windowed/full screen toggles. This is the best location to create
D3DPOOL_MANAGED resources since these resources need to be reloaded whenever the device is
destroyed. Resources created here should be released in the OnDestroyDevice(). */
HRESULT InitDeviceObjects(LPDIRECT3DDEVICE9 pd3dDevice);

/** This function will be called immediately after the Direct3D device has been reset, which will happen
after a lost device scenario. This is the best location to create D3DPOOL_DEFAULT resources since
these resources need to be reloaded whenever the device is lost. Resources created here should be
released in the OnLostDevice.*/
HRESULT RestoreDeviceObjects();

/** This function will be called at the end of every frame to perform all the rendering calls for the scene,
and it will also be called if the window needs to be repainted. After this function has returned, application
should call IDirect3DDevice9::Present to display the contents of the next buffer in the swap chain*/
HRESULT Render(float fTime);

/** This function will be called once at the beginning of every frame. This is the best location for your
application to handle updates to the scene, but is not intended to contain actual rendering calls, which
should instead be placed in the Render. */
HRESULT FrameMove(float fTime);

/** This function will be called immediately after the Direct3D device has entered a lost state and before
IDirect3DDevice9::Reset is called. Resources created in the OnResetDevice should be released here,
which generally includes all D3DPOOL_DEFAULT resources. See the "Lost Devices" section of the
documentation for information about lost devices. */
HRESULT InvalidateDeviceObjects();

/** This callback function will be called immediately after the Direct3D device has been destroyed, which
generally happens as a result of application termination or windowed/full screen toggles. Resources
created in the OnCreateDevice() should be released here, which generally includes all
D3DPOOL_MANAGED resources. */
HRESULT DeleteDeviceObjects();

/** This function should be called only once when the application end, one can destroy game objects
here.*/
HRESULT FinalCleanup();

/** process game input.*/
void HandleUserInput();

/** Before handling window messages, application should pass incoming windows messages to the
application through this callback function.*/
LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam );

protected:
// ... members here

```

```
};
```

CParaEngineApp does not create the Win32 window, but it need to know the current window handle associated with the render device; neither does it create the Direct3D devices, instead it saves the device pointer and use it to draw 3D objects. Although we will not create a game engine that supports multiple render device types, it is good practice to use a simple wrapper for our device object. Because there is usually only one valid device at any given time, we can use a global singleton class to store the device object. We will write a class called CGlobal and save the device object in it. In future, we will put other global objects that can only have one instance in the game engine to it, such as the scene manager, the GUI engine, the global terrain object, etc. The following is an example of CGlobal class.

```
/** prestore device parameters for the DirectX device.*/
struct DirectXEngine
{
    LPDIRECT3D9      m_pD3D;           // The main D3D object
    LPDIRECT3DDEVICE9 m_pd3dDevice;     // The D3D rendering device
    D3DCAPS9         m_d3dCaps;        // Caps for the device
    D3DSURFACE_DESC  m_d3dsdBackBuffer;
    ... // others members and functions
}

// forward declarations
...
class CGlobals{
public:
    static IDirect3DDevice9 * GetRenderDevice();
    /** get DirectX engine parameters */
    static DirectXEngine& GetDirectXEngine();
public:
    static CSceneObject* GetScene();
    static SceneState*GetSceneState();
    static CPhysicsWorld* GetPhysicsWorld();
    static CParaWorldAsset* GetAssetManager();
    static CAISimulator* GetAISim();
    static CEnvironmentSim* GetEnvSim();
    static CGUIRoot* GetGUI();
    static CReport* GetReport();
    static CFileManager* GetFileManager();
    static CGlobalTerrain* GetGlobalTerrain();
    static COceanManager* GetOceanManager();
    static float GetGameTime();
    static HWND GetAppHWND();
    static const D3DXMATRIX* GetIdentityMatrix();
};
```

Whenever we need to use the device object, we can call CGlobal::GetRenderDevice() or CGlobal:: GetDirectXEngine() for more pre-stored information about the device. Please note that CGlobal is not a formal singleton class, it is actually a bunch of static functions. When using it, it looks like a namespace called CGlobal.

Some people may argue that global members and functions are not good in C++ and they should be passed as parameters instead. The reason that global objects are used quite often in a game engine is given below.

- Global objects do not need to be passed as parameters. Many source files in the game engine will use these global objects, and they do not need to include them in the parameter list.
- It makes other C++ header files clean, i.e. without the ugly name like IDirect3DDevice9* and it also saves many forward declaration lines in the header files.

However, global objects should not be used without consideration. The above code already lists all major global objects that will be used in the entire game engine.

2.3.3 Time Controller

In ParaEngine, several global timers are used to synchronize engine modules that need to be timed. Figure 2.3 shows a circuitry of such modules running under normal state. The darker the color of the module is, the higher the frequency of its evaluation.

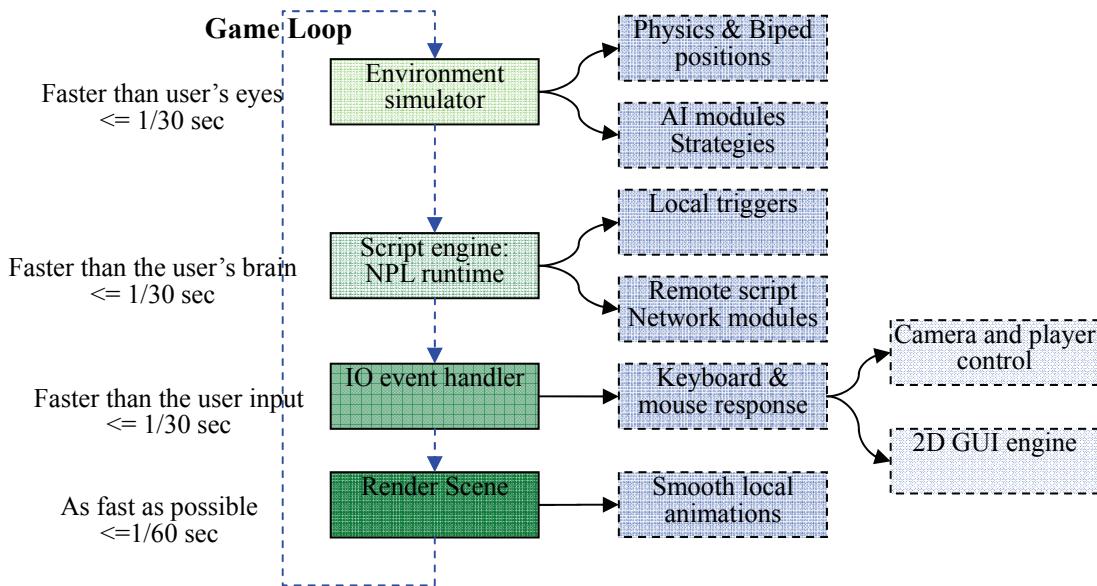


Figure 2.3 Timing and I/O in ParaEngine

One of the first jobs that a game loop does is to decide how often a certain module should be called and the actual and/or logical time delta between two successive calls. E.g. the Render() function may be called 30 times per second, with a time step of approximately 1/30 second.

Time management is an advanced topic. We will cover it by an entire chapter. But so far, let us just do a most simple time controller which outputs actual (unprocessed) time step. In ParaEngine, we call it frame rate controller.

```

class CFrameRateController
{
public:
enum ControllerType{
    FRC_NONE = 0, // output delta time without interpolation
    FRC_CONSTANT, // output a constant delta time disregarding the input time
    FRC_CONSTANT_OR_ABOVE, // output a constant delta time or a smaller one
    FRC_FIRSTORDER, // interpolating using a linear function
    FRC_SECONDORDER // interpolating using a second order function
}

```

```

};

ControllerType m_nType;/// Frame Rate Controller Type
float m_fTime; // current time
// other members and functions
...
float FrameMove(float time){
    switch(m_nType){
        case FRC_NONE:
            float fElapsedTime = fTime - m_time;
            m_fTime = time;
            return fElapsedTime;
    }
}

```

In the game loop, we can use the frame rate controller class as below. Please note that we instantiate the frame rate controllers as global (static) objects. Different modules of the game engine may depend on different time controllers with different parameters.

```

static CFrameRateController g_simFRC;
static CFrameRateController g_renderFRC;
Game loop (){
    fTime = GetTime (); // better use QueryPerformanceFrequency() internally.
    call Framemove(fTime)
    call Render(fTime)
}
Framemove(float fTime){
    float fTimeDelta = g_simFRC.FrameMove(fTime);
    ... TODO
}
Render(float fTime){
    float fTimeDelta = g_renderFRC.FrameMoveDelta(fTime);
    ... TODO
}

```

To get the current time, it is advised to use the more advanced version of time functions provided by the operating system, such as `QueryPerformanceFrequency`, or at least `timeGetTime()`.

2.3.4 Log System

Log system is used to output messages during the program execution. The basic log function opens a file, appends text to it and then closes the file.

```

void CLog::AddLogStr(const char * pStr)
{
    if (pStr==NULL) {
        return;
    }
    if (gLogFileHandle==NULL) {
        gLogFileHandle = fopen(sLogFile.c_str(),"w");
    }

    if(gLogFileHandle){
        fprintf(gLogFileHandle, pStr);
        fflush(gLogFileHandle);
    }
}
void CLog::WriteFormated(const char * zFormat,...)
{

```

```

va_list args;
va_start(args, zFormat);
_vsnprintf(buf_, MAX_DEBUG_STRING_LENGTH, zFormat, args);
va_end(args);
CLog::AddLogStr(buf_);
}

```

Programmers can write many versions of the log function, such as providing formatted version that optionally outputs file and line number, etc. Some people like to use it with macros, so that log behavior may be different for debug and release build. The book's website contains source code of a log system and a performance analyzer used in our game engine.

Log system can be very useful for debugging as well as game development. A game engine will need to deal with tens of thousands of files, and be fault-tolerance to script and missing files. Logging proves to be useful not only for programmers, but also for game content developers to trace the problems. Hence, it is good practice to output all warnings and errors through the log interface.

2.3.5 Memory Leak Detection

Memory leak is one of the common C++ problems that could be traced by employing some debugging measures. In our game engine, we use the recommended operator new and delete for dynamic memory allocation and de-allocation. Undiscovered memory leak problems in a computer game engine can deteriorate its performance easily. Memory leak usually results from unmatched new and delete calls. There is a trick in C++ which replaces the traditional new and delete operators with a debug version that detects unmatched calls. The system already has these functions in <crtDBG.h>. All we need to do is to include this header as shown below.

```

#pragma once

// VC++ uses this macro as debug/release mode indicator
#ifndef _DEBUG
// Need to undef new if including crtDBG.h which redefines new itself
#define new
#define __new
#endif

// we need this to show file & line number of the allocation that caused
// the leak
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#ifndef _CRTBLD
// Need when builded with pure MS SDK
#define _CRTBLD
#endif

#include <crtDBG.h>

// this define works around a bug with inline declarations of new, see
// http://support.microsoft.com/support/kb/articles/Q140/8/58.asp
// for the details
#define new new( _NORMAL_BLOCK, __FILE__, __LINE__)

```

```
#endif
```

2.3.6 Using Mini-Dump

When your application crashes either in release or debug build, we can generate a mini-dump file, which can usually accurately bring developers to the line of source code that caused the crash. Please refer to the Direct X SDK technical article for details.

2.3.7 File Manager

A unified file interface is very useful in a computer game engine. A game engine usually contains tens of thousands of resource files, most of which are read-only. In recent days, it is common to selectively compress and put large number of files in a single package file, such as in a zip archive. This will save disk space, allow multiple versions of a file to coexist, and increase file searching speed. The details of the file system will be covered in an entire chapter. But in this chapter, we will develop the basic file interface and an implementation using the traditional operating system's file API.

In ParaEngine, most resource files are loaded through the CParaFile interface, as given below.

```
class CParaFile{
    FileHandle m_handle;
    bool eof :1;
    bool blsOwner:1;
    bool bDiskFileOpened:1;// whether a disk file handle is opened
    char *buffer;
    size_t pointer,size;
    string m_filename;
public:
    CParaFile();
    /** Open a file for immediate reading.*/
    CParaFile(const char* filename);

    /** no longer release the file buffer when this file object is destroyed. */
    void GiveupBufferOwnership();

    /** read byte stream from file at its current location. The buffer pointer will be advanced.*/
    size_t read(void* dest, size_t bytes);
    size_t getSize();
    size_t getPos();
    char* getBuffer();
    char* getPointer();
    bool isEof();
    void seek(int offset);
    void seekRelative(int offset);
    void close();
}
```

The above is a primitive file interface with read-only access. It is easy to implement. The interesting part of this special implementation is that when the file object is created, it reads the entire file to a memory buffer, and when the file object is released it also releases the buffer. However, it allows the user to keep the buffer by calling GiveupBufferOwnership().

To use the file interface, e.g. opening a texture bitmap file, one can use the following code:

```
void LoadTexture(){
    CParaFile myFile("c:\\picture.bmp");
    char* buffer = myFile.getBuffer();
```

```

if(buffer != 0){
    ... // use the buffer
}
// no need to close the file or release the file buffer.
}

```

2.3.8 Asset Manager

A computer game engine needs to deal with tens of thousands of game assets, such as textures, meshes, sounds, etc. It is time consuming to preload them in to memory. ParaEngine's asset management uses lazy-loading scheme, i.e. when an asset is created, it is not initialized until it is used for the first time. Of course, we can force initializing an asset, but using the lazy-loading scheme can make us load big continuous game scenes without noticeable real-time delays.

Let us first develop a common base class called AssetEntity. In ParaEngine, other asset entities such as texture and mesh are all derived from this base class. In this book, we call resource objects entity, and call scene objects object. For example, a mesh object may need a mesh entity and several texture entities to render it in the scene. We will make a clear distinction of object and entity in the next section. The following code is AssetEntity implementation.

```

/** Base class for managed asset entity in ParaEngine. We allow each entity to have one name shortcut
of string type, so that the entity can be easily referenced in script files through this name. we allow each
asset to be associated with only one name, if multiple names are assigned to the name entity, the latest
assigned names will override previous names. This could be limitation, and will be removed in later
version. However, in the game engine runtime, scene node stores asset entity as pointer to them. Entity
is by default lazily initialized, unless they are called to be initialized. */
struct AssetEntity {
private:
    /** reference count of the asset. Asset may be referenced by scene objects. Once the reference
    count drops to 0, the asset may be unloaded, due to asset garbage collection. however, the asset is
    not completely removed. Pointers to this object will still be valid. only resources (such as texture,
    mesh data, etc) are unloaded.*/
    int mRefCount;
public:
    enum AssetType {
        base=0,
        texture=1,
        mesh=2,
        multianimation=3,
        spritevertex = 4,
        font=5,
        sound=6,
        mdx=7,
        parax=8,
        database=9
    };
    virtual AssetType GetType(){return base;};

    /** this is the unique key object. */
    AssetKey m_key;

    /** whether this entity is initialized;Entity is by default lazily initialized */
    bool blsInitialized:1;

    /** whether this is a valid resource object. */
}

```

```

bool    m_bIsValid:1;

/** whether this is a valid resource object. An invalid object may result from a non-exist resource file.*/
bool IsValid() { return m_bIsValid; };

/** add reference count of the asset. One may later order the asset manager to remove all asset whose
reference count has dropped to 0 */
void addref() { ++mRefCount; }

/** decrease reference count of the asset. One may later order the asset manager to remove all asset
whose reference count has dropped to 0.
@return : return true if the the reference count is zero after decreasing it*/
bool delref() { return --mRefCount<=0; }

/** get the reference count */
int GetRefCount() { return mRefCount; }

AssetEntity():m_bIsValid(true),bIsInitialized(false),mRefCount(0) {}
AssetEntity(const AssetKey& key):      m_bIsValid(true),bIsInitialized(false),mRefCount(0),m_key(key)
{}
virtual ~AssetEntity(){}
/** call this function to safely release this asset. If there is no further reference to this object, it will
actually delete itself (with "delete this"). So never keep a pointer to this class after you have released it.*/
virtual void Release();

/** return the key object. */
AssetKey& GetKey() { return m_key; }

virtual HRESULT InitDeviceObjects(){bIsInitialized =true;return S_OK;};
virtual HRESULT RestoreDeviceObjects(){return S_OK;};
virtual HRESULT InvalidateDeviceObjects(){return S_OK;};
virtual HRESULT DeleteDeviceObjects(){bIsInitialized =false;return S_OK;};
/** Clean up additional resources. This function will only be called before the destructor function.*/
virtual void Cleanup(){};

/** load asset. This function will only load the asset if it has not been initialized. Since ParaEngine uses
lazy loading, it is highly advised that user calls this function as often as possible to ensure that the asset
is active; otherwise, they may get invalid resource pointers. Some derived class will call this function
automatically, during resource pointer retrieval function. E.g. During each frame render routine, call this
function if the asset is used.*/
void LoadAsset(){
    if(!bIsInitialized){
        InitDeviceObjects();
        RestoreDeviceObjects();
        bIsInitialized = true;
    }
}
/** unload asset. Normally this function is called automatically by resource manager. So never call this
function manually, unless you really mean it. */
void UnloadAsset(){
    if(bIsInitialized){
        InvalidateDeviceObjects();
        DeleteDeviceObjects();
        bIsInitialized = false;
    }
}
/** if its reference count is zero, unload this asset object. any reference holder of this object can call this
function to free its resources, if they believe that it will not be needed for quite some time in future.*/
void GarbageCollectMe(){
}

```

```

if(mRefCount<=0){
    UnloadAsset();
    mRefCount = 0;
}
};

```

Each asset entity is associated with a key which uniquely identifies the entity with others of the same type. It has an internal attribute which marks whether the object has been initialized. An entity is defined to be initialized if the entity is loaded from disk file and ready to be used by the render device. It has two pairs of virtual functions, which can be called to initialize/delete or restore/validate the asset entity. It also maintains a reference count for each asset instance. It is a counter to record how many objects are using this asset entity. For example, if there are two mesh objects that reference the same texture entity, then the reference count of the texture entity should be two.

The following program shows the Texture and Mesh asset entity which derives from the base entity class. Please note that for simplicity, we use the class CDXUTMesh which can be found in the DirectX9.0c SDK sample for the mesh entity. In future, you may replace it with one of your own class that parses the mesh data file and renders the mesh.

```

struct TextureEntity : public AssetEntity{
private:
    /// DirectX texture object
    LPDIRECT3DTEXTURE9 m_pTexture;
    TextureInfo* m_pTextureInfo;
public:
    virtual AssetEntity::AssetType GetType(){return AssetEntity::texture;};
    /// the texture can be initialized from an image file
    string sTextureFileName;

    virtual HRESULT InitDeviceObjects();
    virtual HRESULT RestoreDeviceObjects();
    virtual HRESULT InvalidateDeviceObjects();
    virtual HRESULT DeleteDeviceObjects();

    TextureEntity(const AssetKey& key);
    TextureEntity();
    virtual ~TextureEntity();
    LPDIRECT3DTEXTURE9 GetTexture(){
        LoadAsset();
        return m_pTexture;
    };
    ... // other members and functions omitted.
};

struct MeshEntity : public AssetEntity{
private:
    /// For simplicity, use the mesh class provided by DirectX SDK sample
    CDXUTMesh m_mesh;
public:
    virtual AssetEntity::AssetType GetType(){return AssetEntity::mesh;};
    /// name of the x file name holding the mesh object
    string sMeshFileName;

    /// the view culling object used for object-level culling when rendering this mesh Entity.
    /// Generally it contains the bounding box of the mesh.
    CViewCullingObject m_pViewClippingObject;
};

```

```

MeshEntity(const AssetKey& key):AssetEntity(key) {}
virtual ~MeshEntity(){};

virtual HRESULT InitDeviceObjects();
virtual HRESULT RestoreDeviceObjects();
virtual HRESULT InvalidateDeviceObjects();
virtual HRESULT DeleteDeviceObjects();
CDXUTMesh* GetMesh(){
    LoadAsset();
    if(!isValid()) { return &m_mesh; } else { return NULL; }
};

/** init the asset entity object. compute and save the bounding box of the mesh*/
void Init(const char* sFilename);
... // other members and functions omitted.
};

```

To use the TextureEntity and MeshEntity, we simply call GetTexture() and GetMesh() which will initialize the entity if not done so yet.

ParaEngine have many other entity types given in Figure 2.4. But for our skeleton engine, we only need to implement the texture and mesh entity to some extent so far.

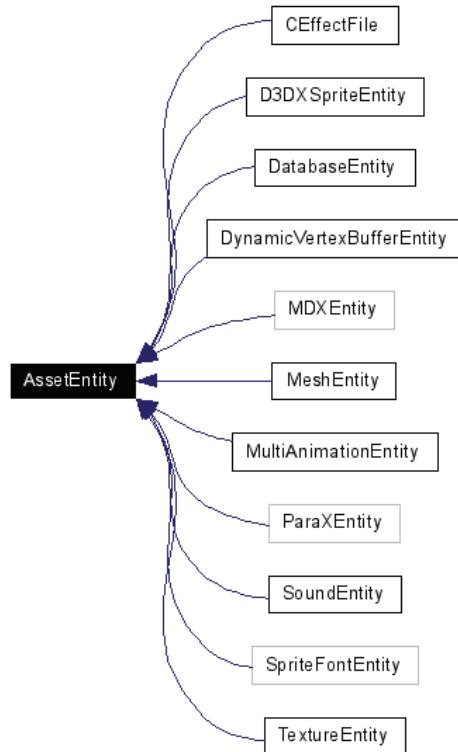


Figure 2.4 Asset entities in ParaEngine

So far, we have shown how individual texture and mesh is wrapped in the unified AssetEntity class. However, in the game engine, we will need to manage thousands of game resource entities. We need a central place to create, store and manage all kinds of asset entities. We can use a C++ template class called AssetManager which keeps a certain kind of asset entities in a hash table (using the STL set template), so that we can locate an asset entity by its key in

$\log(n)$ time. The canonical name of the asset file is usually chosen as the asset key. However, a name shortcut can be optionally assigned to an asset entity, so that it can be easily referenced in script files via this short name. This can be done by using a STL map object to index asset entities by its short name string. Only one short name can be associated with an asset, if multiple names are assigned to the same entity, the latest assigned name will override previous ones. The template code for AssetManager is too long to be included here; one can refer to the source code published on our website. Below is the code outline.

```
/** AssetManager manages a set of asset entities of a certain type. IDTYPE must be AssetEntity derived
class. */
template <class IDTYPE>
class AssetManager {
public:
    struct Compare_less{
        // functor for operator<
        bool operator()(const AssetEntity* _Left, const AssetEntity* _Right) const {
            // apply operator< to operands
            return (_Left->m_key.Compare(_Right->m_key)<0);
        }
    };
    /** human readable name of asset object, which can be used as short cut to retrieve entity from its
    manager. Internally we use the key object to uniquely identify an asset. Short cut can be nil "".*/
    std::map<std::string, AssetEntity*> m_names;

    /** A set of all asset entities.*/
    std::set<AssetEntity*, Compare_less> m_items;

    void Cleanup ();
    virtual void DeleteEntity (AssetEntity *entity)
    void DeleteByName (std::string name)
    /** Check if there is a object with a specified name, and return the pointer to it.*/
    AssetEntity * get (std::string name)
    AssetEntity * get (const AssetKey &key)
    /** Create a new entity object and add it to the manager.*/
    pair< IDTYPE *, bool > CreateEntity (const string &name, const AssetKey &key)
    /** get the entity by its string name */
    IDTYPE * GetEntity (const string &name)

    void LoadAsset ()
    void UnloadAsset ()
    void GarbageCollectAll ()
    virtual void InitDeviceObjects ()
    virtual void RestoreDeviceObjects ()
    virtual void InvalidateDeviceObjects ()
    virtual void DeleteDeviceObjects ()
    ... // other members and functions omitted.
}
```

Finally, we use another singleton class called CParaWorldAsset to store all asset managers for different kinds of assets as below.

```
class CParaWorldAsset
{
    AssetManager <MeshEntity> m_MeshesManager;
    AssetManager <TextureEntity> m_TexturesManager;
public:
    MultiAnimationEntity* GetAnimation(const string& sIdentifier);
    TextureEntity* GetTexture(const string& sIdentifier);
```

```

MeshEntity* LoadMesh(const string& sIdentifier,const string& sMeshFileName);
TextureEntity* LoadTexture(const string& sIdentifier,const string& sTextureFileName, _SurfaceType);
... // other members and functions omitted.
}

```

In the section “Writing the Game Loop”, we have shown how to retrieve global singleton object in the CGlobal class. Thus, to load a mesh, we call

```
AssetEntity* pAssetEntity = CGlobals::GetAssetManager()->LoadMesh(strAssetName, strFilePath);
```

2.3.9 Scene Manager

The entire 3D game world in ParaEngine is composed of two sets of objects. The first set is called scene objects and the second set is called asset entities. The scene objects have little to do with graphics; it is usually just a mathematical presentation of an object in the game world, such as the size and position of a character. A scene object usually does not render itself directly. Instead, it is usually associated with one or several asset entities. These asset entities are usually textures, mesh and animation data that contains the actual artwork for rendering and methods of drawing them.

Throughout this book, we will create several kinds of scene objects. In this section, we will write the root scene object and the mesh object. The root scene object is just a container of various kinds of objects in the game scene, such as the global terrain, the camera, the global bipeds, the physics world, the asset manager, the render states, the sky boxes, the quad tree terrain tiles for holding all 3D scene objects, the AI simulator, and 2D GUI root, etc. But so far, it only needs a camera object and a list of mesh objects for our skeleton game engine.

Most scene objects are derived from a common base class called CBaseObject. See the code below. The actual base object class contains far more functions than in the given code.

```

class CBaseObject{
    enum _SceneObjectType{
        BaseObject=0,
        MeshObject=2,
        SceneRoot=6,
        ... // others
    };
    virtual CBaseObject::_SceneObjectType GetType(){return CBaseObject::BaseObject;};
    string      m_sIdentifier; // unit name used in the scripting language
    ObjectType  m_objType;   // type of the object
    ObjectShape m_objShape;  // shape of the object

    /// object shape parameters
    ...

    list<CBaseObject*>  m_children;
    list<ObjectEvent>    m_objEvents;
public:
    CBaseObject(void);
    virtual ~CBaseObject(void);
    const std::string& GetName();
    void SetMyType(ObjectType t);
    ObjectType GetMyType();

    virtual void GetPosition(D3DXVECTOR3 *pV);
    virtual void SetPosition(const D3DXVECTOR3 *pV) {};
    virtual void GetOBB(CShapeOBB* obb);
    virtual void GetAABB(CShapeAABB* aabb);
}

```

```

/// used as KEY for batch rendering
virtual AssetEntity* GetPrimaryAsset(){return NULL;};

virtual CBaseObject* GetViewClippingObject() { return this;};

virtual void Animate( double dTimeDelta );
virtual HRESULT Draw( SceneState * sceneState);

...// many virtual functions and members omitted here
}

```

The base object keeps the object identifier, object type, a list of child objects and a number of virtual functions, etc. It has two important methods draw() and animate(). The draw method render the object (not including its children), and the animate method advance the animation of the object by a time delta if it contains animation.

Both our root scene object (CSceneObject) and the mesh object (CMeshObject) are derived from the base class. CMeshObject has a mesh entity class member. The detailed implementation of these two classes can be found in our website.

2.3.10 Scripting System

Before we go into the depth of a scripting system in **Chapter 16**; let us first set up a basic scripting system with Lua and Luabind in the skeleton game engine. There are many choices for game scripting system implementation, such as Lua, Python, etc.

The scripting system or NPL runtime environment in ParaEngine is built on top of Lua.

Lua is a popular light-weighted extensible language. It offers good support for object-oriented programming, functional programming and data-driven programming.

Luabind is a library that helps you create bindings between C++ and Lua. It has the ability to expose functions and classes, written in C++, to Lua. For example, if you have a C++ function that creates a mesh object as shown in the left, Luabind will automatically generate code at compile time to register a Lua function, so that one can immediately use the same function in a script file. The binding code in C++ looks magical, and is implemented utilizing C++ template meta programming.

C++ code	Script code
<pre> /** a scene object for scripting*/ struct ParaObject { public: CBaseObject* m_pObj; // a pointer to the object ParaObject(CBaseObject* pObj):m_pObj(pObj) {}; string GetName(){return m_pObj->GetName(); }; }; /** a table of static functions */ class ParaScene{ public: static static ParaObject CreateMesh (){ return ParaObject(new CMeshObject()); } }; ... // register using luabind void CNPLRuntimeState::LoadHAPI_SceneManager() </pre>	<pre> -- create a mesh and print its name local obj = ParaScene.CreateMesh(); local tmp = obj:GetName(); if (tmp == "") then io.print("object has no name"); else io.print(tmp); end </pre>

```

using namespace luabind;
lua_State* L = GetLuaState();
module(L)
[
    namespace_("ParaScene")
    [
        class_<ParaObject>("ParaObject")
            .def(constructor<>())
            .def("GetName", &ParaObject::GetName),

        def("CreateMesh ", &ParaScene:: CreateMesh)
    ]
];
}

```

A fairly complete implementation can be found at our website. Whenever we add new functions to the game engine, we might consider expose them through the scripting interface. Functionalities can be organized in tables. In the above example, ParaScene is a table. In the future, it will include all functions related to scene objects. ParaObject is a data object in script; it could represent any scene object derived from the base object class.

Throughout the design of ParaEngine, we will have the following tables and data objects in the scripting interface (see Table 2.1 NPL Scripting Interface Overview).

Table 2.1 NPL Scripting Interface Overview

Table Name	Description
ParaScene	ParaScene contains a list of functions to create and modify scene objects
ParaAsset	ParaAsset contains a list of functions to manage resources (asset) used in game world composing, such as 3d models, textures, animations, sounds, etc.
ParaUI	ParaUI contains a list of functions to create user interface controls, such as windows, buttons, as well as event triggers.
ParaCamera	The camera controller.
ParaMisc	Contains miscellaneous functions.
NPL	NPL foundation.

Object Name	Description
ParaObject	It represents a scene object.
ParaCharacter	It represents a character object.
ParaAssetObject	It represents an assert entity
ParaUIObject	It represents a GUI object

2.4 Summary and Outlook

This chapter guides you to build a skeleton game engine, which has a potential architecture to be extended to a formal game engine. But it is not mandatory that you write your first game engine using exactly same methods. As we have mentioned at the very beginning of the book, game engine implementation is diverse and subject to constant changes. In the future, the computer will become faster and that the API of the operating system and DirectX, etc will evolve to handle more sophisticated functions internally. Writing a modern computer game engine needs to consider such changes seriously. Lots of traditional algorithms and architectures used in game engines 3 or 5 years ago are losing popularities in today's implementations. The following trends are already changing game engine architecture.

- Multiprocessors: In fact, game platform such as Xbox 360 is already using multiple CPUs and supports parallel programming to some extend.
- GPU: GPU is getting faster and more parallel. Video memory is getting as cheap as system memory. GPU is doing more tasks than its traditional rendering pipeline.
- Hardware accelerated game modules: Other game engine modules besides the graphics module may be hardware accelerated. Physics simulation, for example, are now supported by special hardware. See www.agia.com.
- Internet: game technologies are making the web 3D. It requires game engines to host distributed virtual game worlds. Game engines based on traditional client/server architecture needs evolution.

Finally, we direct you to a number of open source game engines, whose architecture may be useful to you.

- OGRE: very good object oriented game engine framework with a large user community.
- Irrlicht: a good clean implementation of computer game engine.

Chapter 3 Files and File formats

To programmers, a software system can be quickly demystified by examining its protocols and file formats. Even for programmers without former game programming experience, it is quite helpful to glance over all file formats used by a game engine. We not only covers the file formats, but also tell you the average file size, average number of each file types processed by a moderate 3D game, as well as file distributions. Novice programmers and designers can prepare their minds and start reflecting on how large volumes of game data can be managed efficiently at real time. This is a special and independent chapter that we designed to get you very close to a complete computer game engine in a bottom up manner before you even know their implementations.

Please note that this chapter is NOT written so that one can understand everything, but it tries to give the big picture of a computer game engine in a bottom up manner.

3.1 File Format from Game Requirement

The game engine usually does not contain game data. Instead all game content are read from the file system, database or network on the fly. General game data that consumes most memory space is textures, model meshes, animations, terrain elevations, sounds, strings, and scripts. All of them are read by the game engine through the file interface. And there are unlimited ways to format them in files. The basic principle is to avoid data duplications in multiple files.

The file formats that a game engine support usually depends on game requirement. Engine designers should make choices based on a number of requirements.

- Data reuse: if the game engine needs the data in a number of places, then it should be put in a separate file. E.g. if the key frames of skeleton animations should be used by multiple mesh characters, then animation data and mesh data needs to be put in separate files; if a terrain height field is used at multiple places of the terrain surface, then they may also be put in a separate file.
- Performance: By designing new file format that matches the internal data structure of the game engine, we can get high performance and low memory uses. For example, the most effective texture format in DirectX is DDS file format. Please also note that, one does not need to put too much effort on file format performance issues, because the performance gain of many such optimizations are not as noticeable as the programmers might expect.
- Eligibility: XML or other text based encoding is more eligible than binary encoding. Some very good game engine prefers text encoding for models, animations and almost everything except for texture bitmaps, while still having very good performance. The bottleneck of game performance is usually not at how one decodes data in files (in fact, disk file I/O takes more time than that). Some descriptive languages such as XML, DirectX X file, VRML have binary equivalent to their text encoding.
- Internet: For data to be passed in the Internet, it needs to be eligible and ideally self-descriptive. Most importantly, it has to be small or even streamed. So far, very few game engines make this a requirement in their design.

- Protection: By encoding in a proprietary and binary file format, it will increase the difficulty for others to extract game data from the file. The downside is that we will need to design specific tools to build these files from other public file format or software system. For example, we can use the public “tga” or “dds” as the texture file format, or we can slightly modify the file header to create a new texture format, so that users will not easily open these files with a traditional image browser.
- Extensibility: Many old public file formats for describing mesh and animation, etc, lacks extensibility. When we want to add new data fields to a file, it will not be very comfortable with these old file formats. There may be some workarounds, such as utilizing the comment lines supported by the file format; but generally it is not an elegant solution. Some newer file formats have very good extensibility, such as XML and DirectX X file. But sometimes, using them will not be easier than defining a new file format.
- Generality: Most game engines supports multiple file formats for the same game data. For example, most game engine supports two or more public texture file formats and one or more public mesh file formats.

If you are the reader who wants to start a new game engine project, it is worth spending some time on game requirements from the file perspective, and making the decision of which file formats to use or plan to use at an early stage of your development.

3.2 ParaEngine File Formats Overview

This section describes the file format requirement derived from the requirement of ParaEngine. In ParaEngine, we want to build a game world on a continuous and infinitely large terrain; we want to user to be able to manipulate everything in the virtual game world without any noticeable delay in graphics or physics. More specifically, we need the user to create and modify terrain and ocean, to lay and transform objects such as trees and buildings in the game world, and to create live characters and sequences of animations, all in real time. We want to derive a new virtual world from one or several existing game worlds without duplicating data, so that users can publish a large personal virtual world in very small size to the Internet and that large number of people could share a single virtual world with data distributed on each person’s computer. We need game data to be packed in smaller files for network transfer and accumulative display.

From the above requirement listing, we can design the outline of ParaEngine file format. File relationships is given in Figure 3.1. The basic file types are given below.

- World Config file: the entry file for a virtual world; it contains references to a number of terrain config files.
- Terrain Config file: the entry file for a fixed sized square terrain; it contains references to terrain elevation file, base terrain texture files, texture mask files, and on load script, etc.
- Terrain Elevation file: height maps of terrain surface.
- Terrain texture Mask file: the alpha maps for different texture layers on the terrain surface.
- Terrain On load script file: a script file for loading different kinds of scene objects (such as meshes, lights, levels, characters, water, etc) to the game world.

- Mesh & Animation file: 3D meshes and animation data, etc.
- Texture files: all kinds of textures used in the game engines.
- Movie file: a movie sequence of a character.
- Script files: script files in NPL, which are compiled and executed by the NPL runtime environment on the fly.
- Game loop file: user specified script files which are executed repeatedly.
- Log: the text file for log result.
- Effect files: small pieces of code used by GPU.
- Sound files.
- Database file: client side database file for some game specific logics.
- Archive file: a package file that contains a number of compressed or uncompressed files.
E.g. Zip file is a most frequently used archive file.

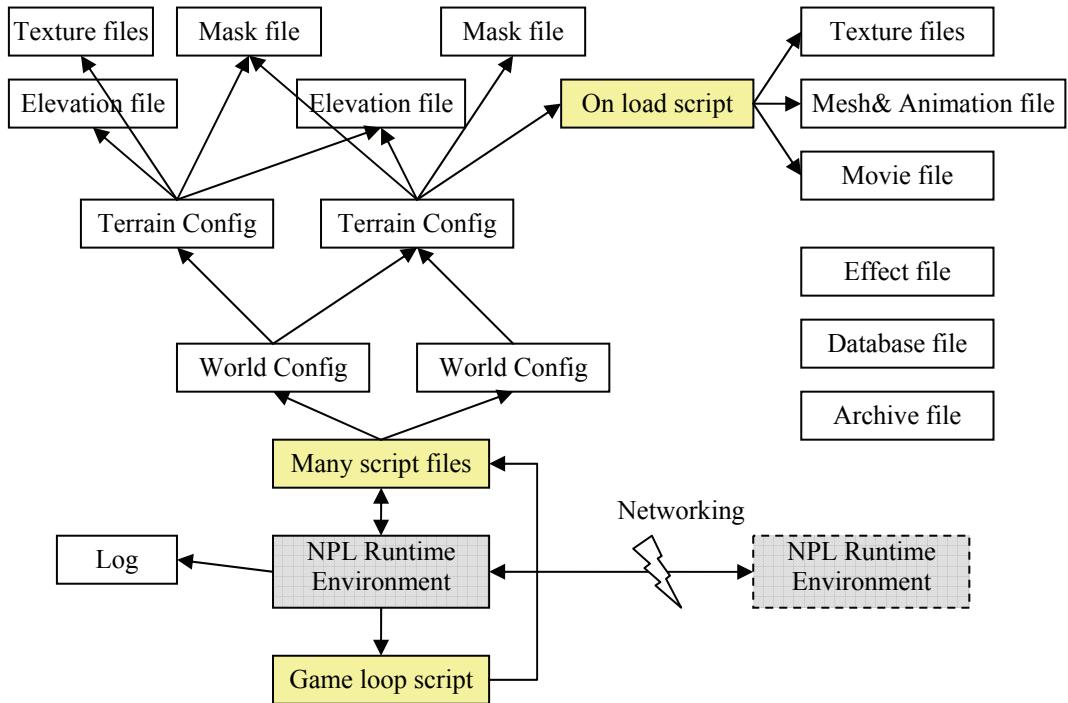


Figure 3.1 File relationships in ParaEngine

Please note that we did not list files used by the NPL network engine and files used in game engine configuration. We will cover network related issues in volume II.

One of the characteristics of this file design is that it breaks game data into many small and reusable files. For a 1600 (km*km) continuous and non-repeated 3D game world filled with forests, villages and cities. The file sizes are approximately given in Figure 3.2. We have

shown both compressed and uncompressed size. But it is the compressed size that we care most. Non-listed files are too trivial in size to be listed.

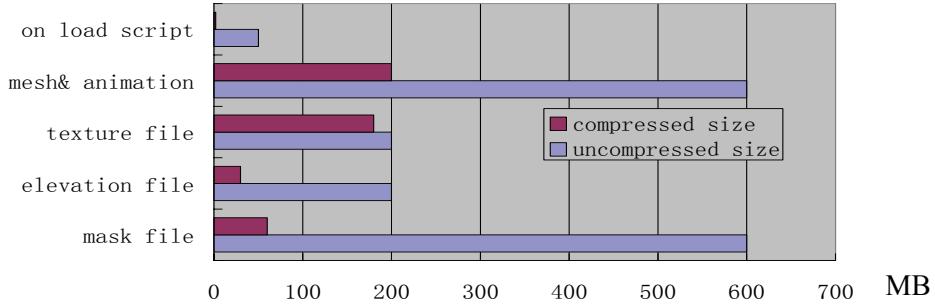


Figure 3.2 File relationships in ParaEngine

The uncompressed size of a single file is given below:

- On load script: approx. 0-100KB.
- Mesh & animation: approx. 10KB-1000KB
- Texture file: 43KB, 86KB (256*256 pixels with alpha), etc
- Elevation file: 64KB
- Mask file: approx. 64KB-1024KB

You can estimate the number of files from the above information. The total files are approx. 10, 000. The data given above could differ greatly from game to game. It just helps engine programmers without actual game development experience to get an impression of the scale of a moderate sized 3D game today.

3.3 ParaEngine File Formats

In this section, we will give the detailed specification of file formats used in the game engine. We use a casual way to describe the file format, so that readers can get the outline of a file format more vividly. Typically, we use `/** x */`, `--x` and `--[[x]]` to denote comment lines; use `[x]` to denote optional field in text encoded file, `{}` for scoping in binary encoding, and we mix actual file examples with their definitions for most text encoded file formats.

3.3.1 World Config File

The game world is partitioned in to many equal sized squares as shown in Figure 3.3. Each square is called a terrain tile, and is specified by a 2D coordinate. Each terrain tile contains a terrain surface, and scene objects which are on that terrain. The world config file is an entry file for a virtual world, defining the attributes of the game world and the distribution of terrain tiles.

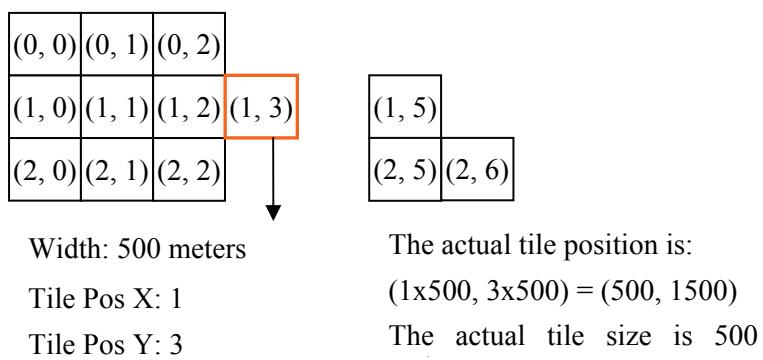
Name: World Config File Specification
Encoding: plain text
Desc: Lattice Terrain Specification. (Pay attention to spaces between characters).It consists of the size

of a terrain tile and a mapping from tile position to their single terrain configuration file. Please note the first line "type = lattice" is mandatory, otherwise it will be mistreated as a single terrain configuration. If there are multiple single configuration files for the same tile position, the first one will be used. If there are multiple tile positions referring the same single terrain configuration file, copies of the same terrain will be created at these tile positions

Notes: The order of tiles is not important.

```
type = lattice
TileSize = 500
-- tile0_0.txt refers to a single terrain configuration file.
(0,0) = terrain/data/tile0_0.txt
(2,2) = terrain/data/tile2_2.txt
(3,3) = terrain/data/tile3_3.txt
(4,2) = terrain/data/tile4_2.txt
```

In a typical world, the number of tiles is 64*64 and the tile size is 500 meters, which could describe a continuous world of 32*32(km.km).



“Sample/sample_1_3.lua”: A loading script file for (1, 3)

Figure 3.3 Grid-based virtual world partition

3.3.2 Terrain Config File

Terrain config file is the entry file for a fixed sized square terrain, which is usually referenced by the world config file. It contains references to terrain elevation file, base terrain texture files, texture mask files, and on load script, etc. On load script contains scene objects on the terrain tile.

Name: Terrain Config File Specification
Encoding: plain text
Desc: A single terrain object can be loaded from a configuration file. the following file format is expected from the file, pay attention to spaces between characters:
Notes: Some fields of the config file are optional. And some fields may be neglected or overridden. E.g. the size of terrain square could be overridden by the one defined in the world config file.
-- The script file to be executed after the terrain is loaded. It is optional. OnLoadFile = script/loader.lua -- the height map file, it can either be a gray-scale image or a Para-RAW elevation file. Heightmapfile = texture/data/elevation.raw -- The main or base texture. this is optional. [MainTextureFile = texture/data/main.jpg] -- The common texture. this is optional. [CommonTextureFile = texture/data/dirt.jpg] -- size of this terrain in meters Size = 512 -- the height value read from the height map will be scaled by this factor.

```

ElevScale = 1.0
-- whether the image is vertically swapped. It takes effects on gray-scale height map image
Swapvertical = 1
-- hight resolution radius, within which the terrain is drawn in detail.this is optional.
[HighResRadius = 30]
-- we will use a LOD block to approximate the terrain at its location, if the block is smaller than
fDetailThreshold pixels when projected to the 2D screen.
DetailThreshold = 9.0
--[[ When doing LOD with the height map, the max block size must be smaller than this one. This will be
(nMaxBlockSize*nMaxBlockSize) sized region on the height map]]
MaxBlockSize = 8
-- the matrix size of high-resolution textures.
DetailTextureMatrixSize = 64
--[[ the terrain holes specified in relative positions. There can be any number of holes following the
"hole". This section is optional.]]
hole
(10,10)
(200,300)
-- number of texture files used in the mask file .
NumOfDetailTextures = 3
texture/data/detail1.jpg
texture/data/detail2.jpg
texture/data/detail3.jpg

```

3.3.3 Terrain Elevation File

The engine will assume that the elevation file is of raw terrain elevation type; otherwise it is treated as a gray scale image. In a gray scale image, a pixel RGB(0.5,0.5,0.5) is of height 0. However, gray scale image has only 8-bits for the height levels, which is insufficient for most game application. That is why we need another high resolution terrain height map file format called Raw Terrain Elevation file, given below.

Name: Terrain Elevation File Specification
Encoding: binary file with file extension ".raw"
Desc: height map of a terrain surface. The content of RAW elevation file is just a buffer of "float[nSize][nSize]", please note that nSize must be power of 2 or power of 2 plus 1. i.e. nSize = (2*...*2) (2*...*2+1); E.g. if the height map is a grid of 129*129, then the uncompressed file size will be 129*129*4 bytes=65KB.
Notes: The file name must end with ".raw", for example. in the single terrain configuration file: Heightmapfile = terrain/data/LlanoElev.raw
FLOAT (4 bytes) FLOAT (4 bytes) ... FLOAT (4 bytes) FLOAT (4 bytes)

3.3.4 Terrain Mask File

Terrain mask file stores the alpha maps for different texture layers on the terrain surface. It describes how multiple layers of textures are blended on the terrain surface. Please Figure 3.4 for a demo of the terrain surface. In the figure, the terrain surface is painted by blending several detailed textures over a base texture. For more information, please see **Chapter 9**.

Name: Terrain Detailed Texture Mask File Specification
Encoding: binary file with file extension ".mask"
Desc: It stores the alpha maps for different texture layers on the terrain surface.
Notes: the name of the mask file is computed as below: the fold of terrain configure file + folder with terrain config file name+.mask"

e.g. if the terrain configure file is "sample/config/sample_0_0.config.txt" then the mask file name will be "sample/config/sample_0_0.mask"
<pre> DWORD nDetailTextures; // number of detailed textures in the texture set of the terrain // following nDetailTextures number of texture string block each of the following format { DWORD nStrLen; // length of the string, do not count the "\0" BYTE str[nStrLen]; the string without the trailing "\0" } DWORD nCellCount; // number of cells , usually 8x8=64 // following nCellCount block of texture cell data, each has the following format { DWORD numDetails; // number of detail textures in the cell // following numDetails blocks of detail texture data, each has the following format { DWORD nSharedIndex; // index into the texture set of the associated terrain. // if nSharedIndex is 0xffffffff, there are no data following this block, otherwise it // is a raw alpha mask in the following format. BYTE MaskData[256*256]; // raw mask data } }</pre>

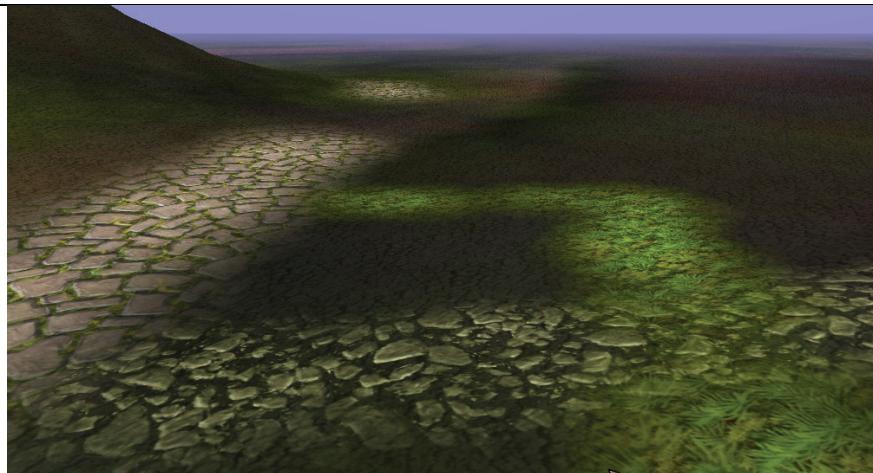


Figure 3.4 Alpha blending of multiple textures on the terrain surface

3.3.5 Terrain On Load Script

Terrain On load script is a script file for loading different kinds of scene objects (such as meshes, lights, levels, characters, water, etc) to the game world. It is referenced by the terrain config file, so that when that terrain tile is loaded to the game world, the game engine can load scene objects that should appear on that piece of terrain. The syntax of the script file is based on Lua, and its function APIs are defined in NPL. We will only give an example here.

Name: Terrain On Load Script
Encoding: script file
Desc: loading scene objects to the scene
Notes:
... (lots of code omitted) -- The following code shows the creation of a static 3D tree on a specified position of the terrain. local asset = ParaAsset.LoadStaticMesh("", "sample/trees/tree1.x") local player = ParaScene.CreateMeshPhysicsObject("",asset, 1,1,1, true, "0.193295,0,0,0,0.187032,-0.0488078,0,0.0488078,0.187032,0,0,0"); player:SetPosition(148,120.156,95);player:SetFacing(0);sceneLoader:AddChild(player);

```
... (lots of code omitted, that create other scene objects)
```

Figure 3.5 shows a 3D scene which is constructed by on load script. Usually an on load script contains the creation code for a few hundreds or over a thousand scene objects in a certain terrain tile.



Figure 3.5 From On Load Script to 3D scenes

3.3.6 Texture File

Usually, a game engine does not need to define its own texture file format. The internal presentation of a general texture in most game engines are the same, which is the default file format consumed by your graphics card. Hence what the game engine does is to convert whatever file formats from the disk to this internal file formats. Some game engine prefers DirectX's DDS file format, because it is the closest format to the internal one. Popular texture file formats are DDS, TGA, BMP, JPG, PNG, etc. A game engine usually supports all of them and specially favours one or two of them. For example, in ParaEngine, we advice all 3D model textures to be in dds file format, and that all 2D UI textures are in png file format.

3.3.7 Mesh and Animation File

Mesh and animation file format is a very flexible format defining mesh and animation data. Because of its flexibility, it is used in a number of places, such as defining static meshes with physics, meshes with level of details, simple animated meshes, characters with complex skeletons and many animations, particle systems, etc. It is the most complex file format in ParaEngine. In other game engines, separate file formats may be used for different kinds of purposes. However, in our implementation, we use a unified file format defined in DirectX's X file format. X file format is an extensible file format which supports both text and binary encoding. Its file structure and extensibility is similar to XML, but is more compact and faster to parse. We use the default X file serializer provided in the DirectX's helper functions. The

file format presented here is optimized for the parser provided in the DirectX9.0c SDK. If one is not familiar with the way DirectX's parser works, it is advised to write one's own file parser.

Name: Mesh and Animation file specification
Encoding: text or binary file defined in DirectX's X file format
Desc: storing mesh and animations.
Notes: Defined in DirectX X file template. The template is registered by the parser for file parsing. Data is in a tree hierarchy as shown below. All nodes are optionally.
-- Overview of the file
<pre> ParaXHeader{...} ParaXBody { [XGlobalSequences{...}] [XVertices{...}] [XTextures{...}] [XAttachments{...}] [XViews{...}] [XIndices0{...}] [XGeosets{...}] [XRenderPass{...}] [XBones{...}] [XCameras{...}] [XLights{...}] [XAnimations{...}] } XDWORDArray "ParaXRawData" {...} -- x file in DirectX retained mode </pre>
-- X File Template
<pre> xof 0303txt 0032 # date: 2006.1.5 template ParaEngine{ <00000000-0000-0000-0000-123456789000> [...] } template ParaXHeader { <10000000-0000-0000-0000-123456789000> DWORD id; array UCHAR version[4]; DWORD type; DWORD AnimationBitwise;# boolean animBones,animTextures Vector minExtent; Vector maxExtent; DWORD nReserved; } template ModelAnimation{ <10000002-0000-0000-0000-123456789000> DWORD animID; DWORD timeStart; DWORD timeEnd; FLOAT moveSpeed; DWORD loopType; </pre>

```

DWORD flags;
DWORD playSpeed;
Vector boxA;
Vector boxB;
FLOAT rad;
}
template AnimationBlock {
<10000003-0000-0000-0000-123456789000>
WORD type;
WORD seq;
DWORD nRanges;
DWORD ofsRanges;
DWORD nTimes;
DWORD ofsTimes;
DWORD nKeys;
DWORD ofsKeys;
}
template ModelBoneDef {
<10000004-0000-0000-0000-123456789000>
DWORD animid;
DWORD flags;
WORD parent;
WORD geoid;
AnimationBlock translation;
AnimationBlock rotation;
AnimationBlock scaling;
Vector pivot;
}
template ModelVertex {
<10000006-0000-0000-0000-123456789000>
Vector pos;
array UCHAR weights[4];
array UCHAR bones[4];
Vector normal;
Coords2d texcoords;
DWORD Color;
}
template ModelView {
<10000007-0000-0000-0000-123456789000>
DWORD nIndex;
DWORD ofsIndex;
DWORD nTris;
DWORD ofsTris;
DWORD nTex;
DWORD ofsTex;
}
template ModelGeoset {
<10000008-0000-0000-0000-123456789000>
WORD id;
WORD vstart;
WORD vcount;
WORD istart;
WORD icount;
Vector v;
}
template ModelTexUnit{
<10000009-0000-0000-0000-123456789000>
WORD flags;
WORD order;

```

```

WORD textureid;
}
template ModelTextureDef {
<1000000d-0000-0000-0000-123456789000>
DWORD type;
DWORD flags;
STRING name;
}
template ModelLightDef {
<1000000e-0000-0000-0000-123456789000>
WORD type;
WORD bone;
Vector pos;
AnimationBlock ambColor;
AnimationBlock ambIntensity;
AnimationBlock color;
AnimationBlock intensity;
AnimationBlock attStart;
AnimationBlock attEnd;
AnimationBlock unk1;
}
template ModelCameraDef {
<1000000f-0000-0000-0000-123456789000>
DWORD id;
FLOAT fov;
FLOAT farclip;
FLOAT nearclip;
AnimationBlock transPos;
Vector pos;
AnimationBlock transTarget;
Vector target;
AnimationBlock rot;
}
template ModelAttachmentDef {
<10000014-0000-0000-0000-123456789000>
DWORD id;
DWORD bone;
Vector pos;
AnimationBlock unk;
}
template ModelRenderPass {
<10000015-0000-0000-0000-123456789000>
WORD indexStart;
WORD indexCount;
WORD vertexStart;
WORD vertexEnd;
DWORD tex;
float p;
WORD color;
WORD opacity;
WORD blendmode;
DWORD order;
DWORD geoSet;
DWORD renderstateBitWise; # usetex2, useenvmap, cull, trans, unlit, nozwrite
}
## ParaXBody contains array blocks
template ParaXBody{
<20000000-0000-0000-0000-123456789000>
[...]

```

```

}

##### array blocks #####
template XVertices {
<20000001-0000-0000-0000-123456789000>
DWORD nType;
DWORD nVertexBytes;
DWORD nVertices;
DWORD ofsVertices;
}
template XTextures {
<20000002-0000-0000-0000-123456789000>
DWORD nTextures;
array ModelTextureDef textures[nTextures];
}
template XAttachments{
<20000003-0000-0000-0000-123456789000>
DWORD nAttachments;
DWORD nAttachLookup;
array ModelAttachmentDef attachments[nAttachments];
array DWORD attLookup[nAttachLookup];
}
template XViews{
<20000005-0000-0000-0000-123456789000>
DWORD nView;
array ModelView views[nView];
}
### XIndices0 only for view0 ###
template XIndices0{
<20000006-0000-0000-0000-123456789000>
DWORD nIndices;
DWORD ofsIndices;
}
template XGeosets{
<20000007-0000-0000-0000-123456789000>
DWORD nGeosets;
array ModelGeoset geosets[nGeosets];
}
template XRenderPass{
<20000008-0000-0000-0000-123456789000>
DWORD nPasses;
array ModelRenderPass passes[nPasses];
}
template XBones{
<20000009-0000-0000-0000-123456789000>
DWORD nBones;
array ModelBoneDef bones[nBones];
}
template XCameras{
<2000000d-0000-0000-0000-123456789000>
DWORD nCameras;
array ModelCameraDef cameras[nCameras];
}
template XLights{
<2000000e-0000-0000-0000-123456789000>
DWORD nLights;
array ModelLightDef lights[nLights];
}
template XAnimations{
<2000000f-0000-0000-0000-123456789000>

```

```

DWORD nAnimations;
array ModelAnimation anims[nAnimations];
}
template XDWORDArray{
<20000010-0000-0000-0000-123456789000>
DWORD nCount;
array DWORD dwData[nCount];
}

```

3.3.8 Archive File

Archive file is a package file that contains a number of compressed or uncompressed files. E.g. Zip file is a most frequently used archive file. When a game is released, people like to put all files in several archive files by file type or file usage. Big package file may contain tens of thousands of files. File searching in an archive file is made faster by loading and keeping the entire file directory of the archive file in memory. Below is the ZIP file format illustration, which is a popular archive file format used by many game engines.

Name: Zip File
Encoding: Binary file
Desc: chunks of compressed or uncompressed files. Files are stored in arbitrary order. Large .ZIP files can span multiple diskette media or be split into user-defined segment sizes.
Notes: Zip file is a chunked file format with a directory at the end of the file.

Overall .ZIP file format:

```

[local file header 1]
[file data 1]
[data descriptor 1]
...
[local file header n]
[file data n]
[data descriptor n]
[archive decryption header]
[archive extra data record]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]

```

A. Local file header:

local file header signature	4 bytes (0x04034b50)
version needed to extract	2 bytes
general purpose bit flag	2 bytes
compression method	2 bytes
last mod file time	2 bytes
last mod file date	2 bytes
crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes
file name length	2 bytes
extra field length	2 bytes
file name (variable size)	
extra field (variable size)	

B. File data

Immediately following the local header for a file is the compressed or stored data for the file. The series of [local file header][file data][data descriptor] repeats for each file in the .ZIP archive.

3.3.9 NPL Script File

Scripting will be covered in greater detail in volume II of the book as well as in **Chapter 16** of the book. We will only give some code examples of NPL script, here.

Outline of Script file

```
local function activate()
    if (state == nil) then --state is a global variable
        local playname = "abc";
        -- activate another script on remote machine.
        NPL.activate("ABC: /pol_intro.lua", "state=nil;");
        -- activate another script on local machine.
        NPL.activate("(gl)polintro.lua","");
    else
        ...
    end
end
state={}; -- a global variable (table), usually for passing and share states among NPL files.
NPL.this (activate); --tell NPL which function is used as the activation function of this file.
```

A sample script that creates a “hello world!” dialog

```
local function activate()
    local window, button, text; --declare local variables
    --create a new window called "mainwindow" at (50,20) with size 600*400
    window=ParaUI.CreateUIObject("container","mainwindow","_lt",50,20,600,400);
    --attach the UI object to screen (root)
    window:AttachToRoot();
    --create a new button called "btnok" at (50,350) with size 70*30
    button=ParaUI.CreateUIObject("button","btnok","_lt",50,350,70,30);
    --attach the button to the window
    window:AddChild(button);
    --set text of the button
    button.text="OK";
    --if the button is clicked, delete the window
    button.onclick=[[((gl)script/empty.lua; ParaUI.Destroy("mainwindow"));]];
    --create a new text box called "txt" at (50,50) with size 500*300
    text=ParaUI.CreateUIObject("text","txt","_lt",50,50,500,300);
    --attach the text to the window
    window:AddChild(text);
    text.text="Hello world!!!";
end
NPL.this(activate);
```

3.4 Summary and Outlook

This chapter examines the major file formats used by a computer game engine. When reading the rest of the book, one may frequently go back to this chapter for reevaluations. During the development of a special game, several new file formats may be created. Major updates to file specification can be disastrous, but they are sometimes inevitable. Our suggestion is to prepare for such occasions from the very beginning. Using general and extensible file formats is a wise practice for most occasions.

Table 3.1 is a list of public file formats that one may consider to use in a game engine. All of them have been used successfully in existing games.

Table 3.1 File Formats in Game Engine

File types	Description
DDS, TGA, JPG, BMP, PNG	Textures file formats
X, LWO, M2, 3DS, XML, X3D(VRML)	Mesh and animations
Raw, grey scale image	Terrain Height map
LUA, PYTHON, C sharp(CS)	Script files
Txt, XML	Configuration files
Sqlite (DB), XML	Client database or data set files
Zip, 7zip	Archive files
Wav, mp3, ogg	Sound files
Fx, cg	Shader files

Chapter 4 Scene Management

From this chapter, we will begin our formal journey into computer game engine architecture. We already covered some scene management and asset management in Chapter 1 and 2. In this chapter, we will take a closer look at it.

4.1 Foundation

3D Scene management is about the organization of scene objects, such as static meshes, sky boxes, characters, terrain, ocean, lights, etc. In games, scene objects are vegetations on the terrain, props and buildings, a table, a glass of water, birds flying in the sky, fish swimming in water, creatures lurked in the forest, controllable characters, etc. Some of these scene objects are static; others are mobile. Some of them contain physics, others reacts to physics. Because an engine can not efficiently load and simulate the entire game world, some objects are loaded only when they are potentially visible or related to the current game logics. E.g. in a typical role playing game, only the regions near the active players are loaded to the scene manager.

Scene management is about a dynamic data structure of game objects in the scene, where the graphics rendering pipeline, the physics simulation engine, the game AI modules, etc could quickly get the objects they need for further processing. Most of these modules that use data in the scene manager share a common requirement: that is *querying by spatiality*. E.g. the rendering pipeline wants to get a list of objects near the current camera position, so that objects in the camera view frustum can be rendered; the physics engine wants to know what other objects are in contact with any dynamic object; the AI modules wants to know all other creatures which are within the view radius of any creature. Also, most scene objects, regardless of its internal presentations, can be abstracted by a simpler shape such as box and sphere. This shared property of scene objects are often used by the scene manager for managing the dynamic data structure of game objects.

According to game requirement, there are many ways to organize scene objects. Sometimes, simple structure can be more effective if used properly. For example, if the game is small, such as scroll screen fighter games, one can put everything in a list; if the game map is small, one can put all objects in a grid, i.e. a single two dimensional array. In this book, however, we will only deal with the situation when the game world is too large to be loaded at once and contains too many objects to be traversed linearly in a list. The solution that most game engine adopts involves the use of a tree-based hierarchical data structure.

4.1.1 Hierarchical Scene Management

Instead of covering several solutions, we will only give one hierarchical scene management solution which is used in ParaEngine. Other methods can be found in the outlook section of this chapter.

In our implementation, we use a single root object called scene root object (CSceneObject) as the entry point to the scene, then we organize all scene objects spatially in it. We design a special dummy scene object called terrain tile object (CTerrainTile) for that purposes, the data structure that we use is called *quad-tree*. Here is how it works. At the root of the quad tree is a CTerrainTileRoot object, which represents the entire game world. The root node contains up

to four child terrain tile nodes, which divides the game world into four equal sized square regions; each child terrain tile nodes can contain another four terrain tiles, which further divides the region into four smaller equal-sized square region. Figure 4.1 shows the partition of a game world into gradually smaller square regions.

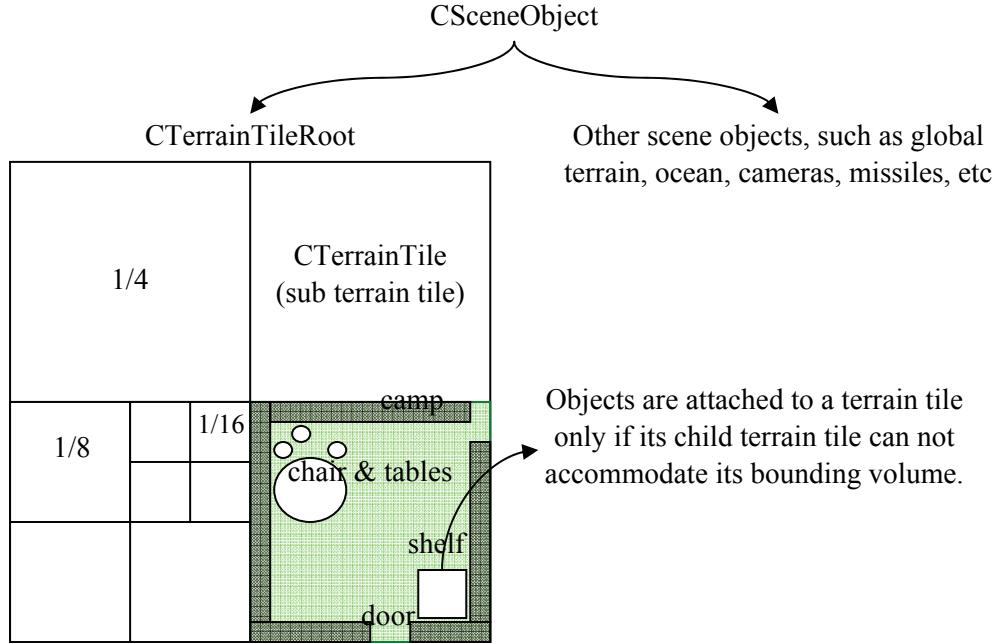


Figure 4.1 Quad-tree of Terrain Tiles

Initially, there is only one terrain tile, i.e. the root terrain tile. As new scene objects are dynamically loaded or added to the scene, a hierarchy of terrain tiles will be created, to which new scene objects are attached. The rule to spawn new terrain tile and attaching scene objects to it is given below.

- Two of the important attributes of the root terrain tile are the size of the game world and the maximum depth of the quad tree. E.g. if the game world is 32000 *32000 (m.m) and the maximum tree depth is set to 8, then the smallest terrain tile is a square of $(32000/2^8=128$ m) in length.
- When the scene is emptied, the quad tree has only a root node without any child terrain nodes.
- When a new scene object is attached to the scene. The bounding volume of the scene object is computed (the basic bounding volume is usually a 3D bounding box which contains the object); the bounding volume is projected to a 2D plane (i.e. ignoring the height of the object).
- The projected bounding volume is tested from the root of the terrain tile. It first tests if the bounding volume falls in to one of the four regions of its child terrain tile. If the bounding volume does not fall into any of the sub regions, the object is attached to the current terrain tile; otherwise we will create the sub terrain tile which contains the bounding volume if it has not been created before and try to attach the new object to that sub terrain tile. This process continues recursively until either the object has been attached to a

terrain tile, or we have reached the maximum depth of the quad tree. In the latter case, the scene object will be attached to the leaf terrain tile anyway.

Figure 4.2 illustrates objects on the hierarchy of quad tree. The quad-tree will automatically expand itself as new objects are attached to it. Although most objects are dynamically inserted into the scene graph tree according to their geographical locations, there are some special objects that do not follow this rule. These objects are active objects that need global simulations. For example, some mobile creatures are attached to the Terrain Tile in which they are patrolling, instead of to the smallest tile that contains its bounding volume.

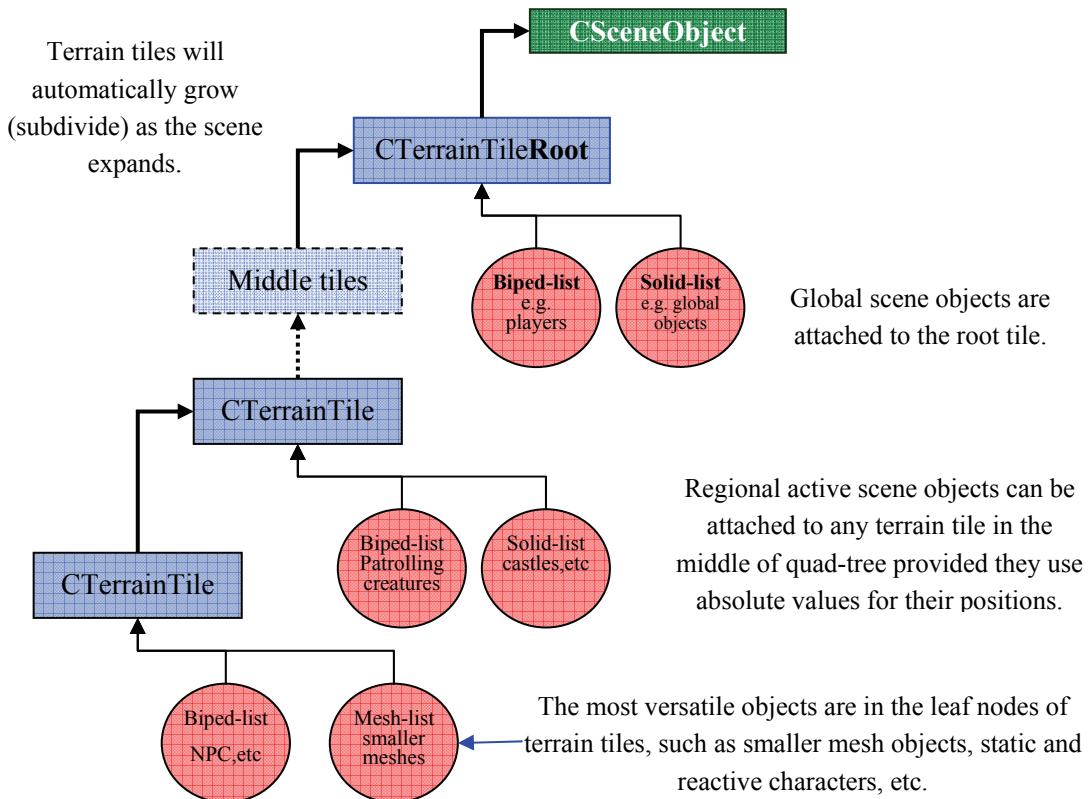


Figure 4.2 Objects on Quad-tree

The reason to use a quad tree as the top level scene hierarchy is that, in a big and extensible 3D world (excluding outer space games), most scene objects are located horizontally in a large plane. Quad tree is thus the most efficient structure to locate a group of objects around a certain 3D position. Please note that quad tree is just one of the many scene organization structures. Other spatial organizational structure includes Octree, BSP tree, Portals, etc, which divides the space according to different rules. They are used elsewhere in the game engine. E.g. a specific object (such as an in-door castle, physics triangle meshes, etc) which is attached to the quad-tree may organize its internal objects (such as sub-geometry, triangles, etc) by these other structures. But the top-level organization of scene objects we advise to use is a quad-tree.

4.1.2 Scene Graph

Some people call scene management, scene graph management. What we have gone through (i.e. the quad-tree terrain tile structure in the scene manager) can also be called a scene graph. A scene graph is a tree where the nodes are scene objects and their sub objects are arranged in some sort of hierarchy. These nodes may be abstract objects, physical objects, meshes, characters, body parts of characters, triangles, etc.

In this book, we sometimes call the scene manager, a scene graph. And when we speak of traversing the scene graph, we really mean accessing objects in the scene in some order.

4.1.3 Using Scene Manager

In previous sections, we talked about how the scene manager organizes scene objects in a tree hierarchy or a scene graph. In this section, we will explore the way to query the scene objects and using the scene manager to complete some specific tasks.

4.1.3.1 View Clipping Object and Bounding Volumes

Before we can efficiently perform queries on the scene graph, we need to design the common base interface for all scene objects on the scene graph. Recall that in Chapter 2, we designed a class called `CBaseObject`, which contains attributes and functionalities that every scene objects should support. Among them there is a function called `GetViewClippingObject()` which returns another, if not the same, `CBaseObject`. The object is a view clipping object that contains the bounding volume information of the scene object. The most commonly used bounding volume presentation is a 3D box. We call a bounding box whose axis are always aligned with the world coordinate, an axis aligned bounding box or **AABB**. We call a bounding box which is arbitrarily positioned in world coordinate system, an oriented bounding box or **OBB**. The view clipping object contains a unified interface for getting information of different bounding volume types from the same object, such as **AABB**, **OBB**, sphere, etc. For example, most static mesh objects' bounding volumes are **OBB**.

Object's bounding volume information is widely used during scene query. For example, if we want to find out if a scene object intersects with another object, we can first tests the two objects by their respective bounding volumes (view clipping objects); if the two view clipping objects do not intersect, we know that the two objects does not intersect neither; if, however, they do intersect, we can continue further testing or simply report contact, depending on our testing requirement. View clipping objects are usually pre-calculated when their associating objects are loaded, so that they can be obtained very fast. See Figure 4.3.

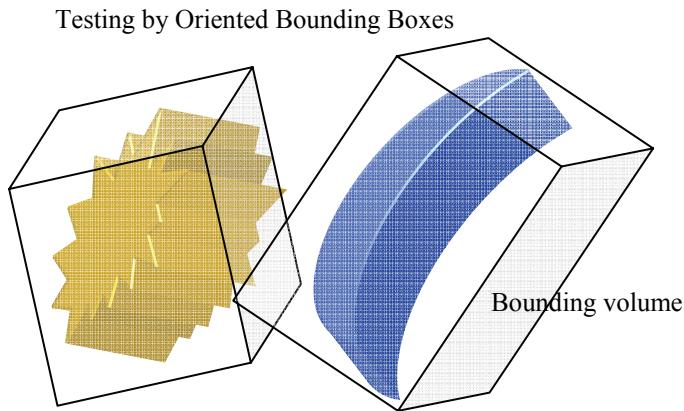


Figure 4.3 Object-level Testing Using Oriented Bounding Boxes

4.1.3.2 Traversing Scene Graph

The most basic use of a scene manager is scene graph traversal. During scene rendering, mouse ray-picking and environment simulation, we will need to traverse the scene graph using different criteria. The goal of a traversal is finding relevant objects such as objects that intersect with the camera view frustum, a ray or a radius. During scene traversal, we may create temporary data structures or even modifying the scene graph, such as the location of an object changes in the scene. The detailed behaviors for different scene traversal tasks are discussed in their respective chapters. Here, we describe the general algorithm of doing a fast scene traversal.

Generally speaking, we can use either breadth-first or depth first traversal, or their combinations. However, we rarely need a full traversal of the scene graph. In most cases, we only want to “touch” objects which intersect with a testing area, such as the camera view frustum, the mouse ray, etc. We can do so by only going deeper into a scene graph level, if and only if it contains or intersects with the testing area. The complete algorithm will be given in the code section.

4.1.4 Dynamic Scene Loading and Unloading

Another issue that is not covered so far is when to load and unload a chunk of scene objects to or from the scene manager. In **Chapter 3**, readers may notice that the game world in ParaEngine is divided into lattices. Each terrain in the lattice is associated with an on load script from which the game engine can construct scene objects in that region. When a user controlled or active character is about to come into a certain terrain lattice, the game engine will automatically load all scene objects in that lattice and the adjacent 8 lattices. To deal with objects spanning the border of two or more lattices and to provide more granular and safe scene loading management, we will introduce another module of the scene manager called managed loader.

A managed loader is a group of scene objects which are usually close to each other. For example, a bunch of small houses can be grouped by one loader; whereas a bigger house with internal decorations can be grouped in another. The loading behavior for all objects in a managed loader is synchronized, i.e. they are attached to the scene manager at one time.

Management loader is also unique in the scene, and that it can only be attached once to the scene manager. With this feature, we can group objects on terrain borders to a unique managed loader and execute the managed loader in all adjacent terrain lattices' on load scripts. Even though, the same managed loader is executed multiple times, objects in the managed loader are only attached once to the scene.

A managed loader is a kind of global scene object for dynamic scene object loading and unloading. The content of a managed loader can no longer be changed once the loader has been attached to the scene. Once objects in managed loader are attached to the scene graph, the ownership of these objects transferred from the loader to the scene manager. The owner of an object is responsible to clean up the object when it is no longer needed. The more exact behavior of a managed loader is given below:

- The objects in a managed loader will be automatically attached to the scene as a single entity.
- Generally speaking, only static objects are put in a managed loader.
- Managed loaders must be identified by a globally unique name.
- If one creates a manager loader with the same name several times, reference to the first loader is returned for all subsequent creation calls.
- The bounding box of a managed loader will be automatically re-calculated as new objects are added to it.
- Linearly traversing all managed loaders in the scene is sufficient to decide which group of scene objects to load or unload. Although this routine is executed when the CPU is free, it is good practice to keep the total number of managed loaders small. Usually a couple of hundred loaders are fine for current hardware for one game scene.
- It is good practice to use managed loaders to attach static scene objects to the scene, instead of attaching them directly. Managed loaders prevent duplicated attachments.
- It is good practice to put static objects which are physically close to each other in a single managed loader.
- It is good practice to put objects on latticed terrain boarders in to a managed loader.

Dynamic scene unloading is not quite an issue in scene management and sometimes not quite necessary. The major penalty for unloaded scene objects is that it uses some system memory for its bounding volumes. But since unused objects are not touched during scene traversal; there are no noticeable penalties for it. Hence, some game engines never remove scene objects from the scene unless it has been manually instructed, such as the user resets the game world. Moreover, in our game engine we separate object from its asset, which further reduced the data keeping size of pure scene object. We do, however, provide a garbage collection function to be manually called to release unused asset entities.

4.2 Scene Manager Architecture

4.2.1 The Scene Root Object

The Scene Root Object (CSceneObject) is a very important class in the game engine. It is the root of the entire scene graph tree. The tree is flat at its root node, but deep on some of its child nodes. For examples, on the root of the scene object is a flat list of engine objects, such as the global terrain, the cameras, the global bipeds, the physics world, the asset manager, the render state, the sky boxes, the quad tree terrain tiles for holding all 3D scene objects, the AI simulator, and 2D GUI root object, etc. The CSceneObject also controls most global states of the 3D scene, such as fog, shadow and some debugging information.

The rendering pipeline is built directly to the root scene object. Actually the entire game world can be accessed through the root scene object. Figure 4.4 shows the Collaboration diagram of the class.

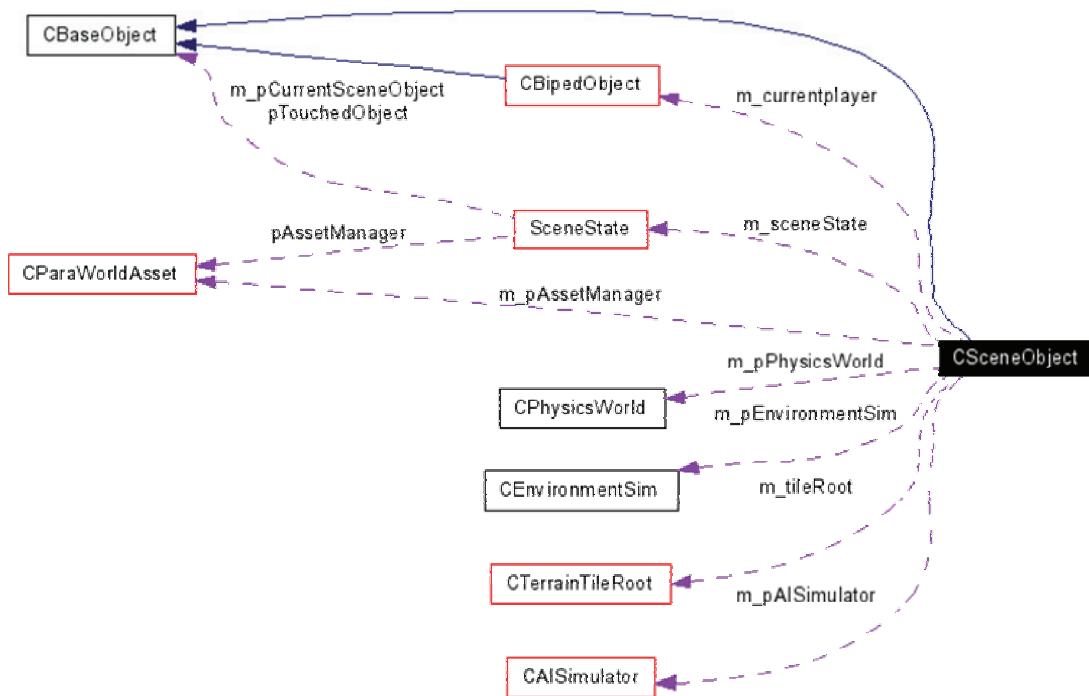
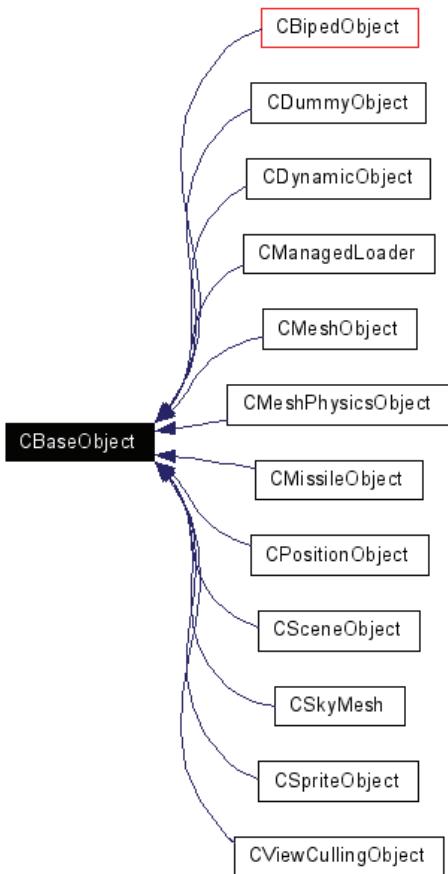


Figure 4.4 The Collaboration diagram for the root scene object

Figure 4.5 is a list of objects inherited from the common scene object interface CBaseObject, which has been explained in [Chapter 2](#). We also list some basic functions related to bounding volume. Bounding volume is used for scene traversal against a testing area.



```

class CBaseObject{
    .. // function related to bounding volume

    void SetBoundingBox(...);
    void GetBoundingBox(...);

    void SetBoundSphere(FLOAT fRadius);
    FLOAT GetBoundSphere();

    /** get the oriented bounding box in world space. One may
     * need to shift by the rendering origin if used for that purposes.*/
    virtual void GetOBB(CShapeOBB* obb);
    virtual void GetAABB(CShapeAABB* aabb);

    virtual bool TestCollisionSphere(...);
    virtual bool TestCollisionObject(CBaseObject* pObj);
    virtual bool TestCollisionRay(...);
    virtual FLOAT GetSphereCollisionDepth(...);
    virtual FLOAT GetObjectCollisionDepth(CBaseObject* pObj);
    float GetObjectToPointDistance(...);

    .. // lots of functions omitted.
}

```

Figure 4.5 The base class and bounding volume functions

4.2.2 Relative Position VS Absolute Position

Although the scene graph is hierarchically structured in a quad tree, scene objects attached to the quad-tree terrain tiles are all specified in world coordinate system. Alternatively, a game engine can use relative position for objects in the scene hierarchy, i.e. a scene object's position is made relative to its parent's position. Although relative position has its merit, we use world coordinates for all independent scene objects. This makes moving objects in a dynamic game world easier. The more important reason is that absolute position works well with other game engine module such as the physics and scripting system. In case of lower level scene hierarchy, relative position is preferred, e.g., body parts of a character object are made relative to their parent node.

4.2.3 Quad Tree Terrain Tiles

The following program shows the interface for terrain tile and the root terrain tile.

```

class CTerrainTile{
public:
    CTerrainTile(void);
    CTerrainTile(float x, float y, float r);
    ~CTerrainTile(void);
    void Cleanup();
}

```

```

public:
    /// -- data structuring
#define MAX_NUM_SUBTILE 4
CTerrainTile* m_subtiles[MAX_NUM_SUBTILE];
list<CBaseObject*> m_listSolidObj;
list<CBaseObject*> m_listFreespace;
/// mobile biped objects that is moving in this region
list<CBipedObject*> m_listBipedVisitors;
/// global name mapping.
map<string, CBaseObject*> m_namemap;

    /// the central position of the terrain.
    float m_fX, m_fY;
    /// the radius of the entire terrain (half the length of the square terrain).
    float m_fRadius;
public:
    /// get object position in the world space
    void GetPosition(D3DXVECTOR3 *pV);
    bool TestCollisionSphere(const D3DXVECTOR3* pvCenter, FLOAT radius);
    bool TestCollisionObject(CBaseObject* pObj);
    float GetSphereCollisionDepth(D3DXVECTOR3* pvCenter, FLOAT radius, bool bSolveDepth);
    float GetObjectCollisionDepth(CBaseObject* pObj);

    int GetSubTileIndexByPoint(FLOAT fAbsoluteX, FLOAT fAbsoluteY);

    CBaseObject* SearchObject(const char* pSearchString, int search_mode=0, int reserved = 0);
    CBaseObject* GetGlobalObject(const string& sName);
    void DestroyObjectByName(const char* sName);
...// some functions and members are omitted
};

class CTerrainTileRoot: public CTerrainTile{
    int m_nDepth;
public:
    /// Reset Terrain
    void ResetTerrain(FLOAT fRadius, int nDepth);
    /// Get and create tile
    CTerrainTile* GetTileByPoint(FLOAT fAbsoluteX, FLOAT fAbsoluteY,
                                FLOAT fPtWidth=0, FLOAT fPtHeight=0);
    /** attach object to the proper tile that best fits it. the terrain tile to which the object is attached is
    returned */
    CTerrainTile* AutoAttachObject(CBaseObject* obj);

    /** detach the object from the tile or its sub tiles. the function will return immediately when the
    first matching object is found and detached*/
    bool DetachObject(CBaseObject * pObject, CTerrainTile* pTile=NULL);
...// some functions and members are omitted
};

```

4.3 Code and Performance Discussion

The code sample for traversing scene graph can be found in the rendering pipeline chapter. The code for performing collision detection between two objects can be found in the picking chapter and physics chapter. The remaining scene manager implementation is just a tree based data structure. We will discuss here about the balance of the quad-tree terrain tile.

4.3.1 Quad-tree Balance Discussion

Quad-tree is used for organizing scene objects spatially. What measures shall we base on for determine the maximum tree depth of the quad tree? The answer to that question usually depends on the density of scene objects and the average volume of scene objects. Ideally, we should minimize the tree depth, while keeping the number of objects in the leaf-node small. Typically, we like to keep the number of objects in the leaf node below 100. The question now becomes how much land 100 objects will usually occupy. In the situation of our game, a 100m*100 m square region might contain 100 top-level objects on average, such as trees and props. So given the size of the entire game world (fWorldSize) and the size of the smallest terrain quad (fSmallestTileSize), in this case, it is 100 m; we can calculate the preferred tree depth with the following formula:

```
// fSmallestTileSize = 100.f;  
int nDepth = (int)(log10( fWorldSize / fSmallestTileSize )/log10(2.f));  
if(nDepth<2) nDepth = 2;
```

4.4 Summary and Outlook

Scene management technique is very dependent on the type of games that a game engine support. For example, typical game types are:

- Indoor games: the game is usually comprised of many independent game levels, each of which takes place in a relatively small region and mainly indoors.
- Outdoor games: the game story takes place in a continuous and large (e.g. tens of kilometers) scene with much longer line of sight. It features large areas of landscape and realistic terrain and ocean rendering. It may also be mixed with indoor scenes. This is the game type our game engine is designed for.
- Outer space games: it features infinitely large space that goes beyond 32 bits floating point presentations and extends in all directions.
- Special games: Any other games that do not belong to the above genres.

Game engines for different game genres are usually different game engines. Of course, a scene manager can be designed to support several scene management methods, but this usually greatly complicates the design of the game engine. This is perhaps one of the reasons why most game engines today only target on one game genre.

The following is a brief introduction of other scene management data structures besides quad tree.

- Binary Space Partitioning (BSP) tree: It is a very efficient way of computing visibility and collision detection in occluded environments. It may include bounding boxes for clipping test. Leafy BSP tree with potentially visible sets (PVS) matrix can do fast occlusion test. The drawback is that BSP only supports static mesh, involves much preprocessing and consumes much maintenance memory.
- Octree (3D) or quad tree (2D): They are very general, scalable and widely applicable data structure for storing objects spatially. It may be weak at occlusion test. Yet with occlusion test supported in hardware, it is sufficient to use this simple data structure in most cases.

- Portal rendering: Like BSP, it allows programmers to quickly decide what is potentially visible in the scene from the camera eye position. It is very suitable for indoor scenes with many dynamic objects in each room. Unlike BSP, the world hierarchy in portal rendering can not be automatically computed, and requires human editing. Visibility is computed in real time, not preprocessed. It is also the most difficult one of the three methods discussed. Moreover, it usually involves additional works for game level editor.

Chapter 5 Rendering Pipeline

In a game engine, the part that deals with graphic storage and displaying is often called 3D engine or rendering engine. The rendering pipeline can be regarded as the main loop in a 3D engine. What the rendering pipeline does is traversing the scene graph, getting objects that are potentially visible by the current camera, and drawing them to the screen. The details of a rendering pipeline can differ greatly for different game engines. This chapter only covers the basics of a rendering pipeline and details about the top level rendering pipeline used in ParaEngine. This top level render pipeline mainly works at the object level and calls the potentially visible scene objects' render function for the actual rendering in polygon level. The pipeline discussed here is suitable for hybrid outdoor and indoor rendering.

5.1 Foundation

5.1.1 Render Pipeline Basics

At one end of the pipeline is polygon data of a scene object in local coordinate system; at the other end is colored pixels in screen coordinate system. The middle stages usually involves, translating the object from local space to world space, check if the object is visible from the current camera settings, remove back and hidden faces if possible, transform polygons from world space to camera space, apply the perspective transform, do the screen space clipping and rasterizing. Formal introduction of a standard rendering pipeline can be found in DirectX SDK. This book will not cover it. If you are a programmer reader, we assume that you already read the rendering pipeline basics in DirectX SDK.

Because a rendering pipeline is a fairly fixed routine, it is accelerated by graphic processing unit (GPU). Since DirectX 8.0, game engine programmers can sidestep the fixed rendering pipeline, and program their own pipeline by replacing some render stage with their own GPU programs. In the coming DirectX Version 10, fixed rendering pipeline is completely replaced by the new programmable pipeline. Today, most 3D engines are based on the CPU / GPU parallel processing architecture. An engine programmer has to program for both and balance the threads of execution.

Figure 5.1 shows the rendering pipeline in CPU / GPU architecture. The white blocks stand for procedures completed by CPU, the colored blocks stand for procedures completed by GPU. The figure has roughly three columns. The left column denotes data set used by pipeline procedures. The middle column denotes pipeline procedures in blocks. The right column contains some brief descriptions of functions completed by a pipeline procedure on the left. The pipeline flows from top to bottom in the figure. It begins by traversing the scene manager and builds a list of potentially visible objects according to the current camera settings. For example, in a large game world, only objects, whose bounding volumes are inside or intersect with the camera view frustum, need to be in the list. A camera view frustum is an enclosed sub space of the game world. We call this procedure object level clipping. This will effectively exclude large number of objects as early in the render pipeline as possible. Then for each objects, the game engine calls its render method which continues the render pipeline downwards. There, the object may need to retrieve polygon and texture data, etc from the asset manager, and then it may optionally do some view culling and view

occlusion tests with the data in the hope to further minimize polygons sent to the next render pipeline stage. Before we pass polygon data (i.e. triangle primitives, etc) down to the GPU pipeline, we must prepare them as formatted buffer objects and setting up the GPU pipeline states and shader programs. After data is sent to GPU, they will be processed in parallel with CPU, following the built-in stages in GPU. Since DirectX 8.0, most stages of the GPU are programmable by engine programmers. The programs executed by GPU are called shader programs. They are mainly a simple function with standard INPUT and OUTPUT format. Since shader programs are executed by GPU in parallel with CPU, they only have access to data in the function parameters, GPU constant tables, and video memory.

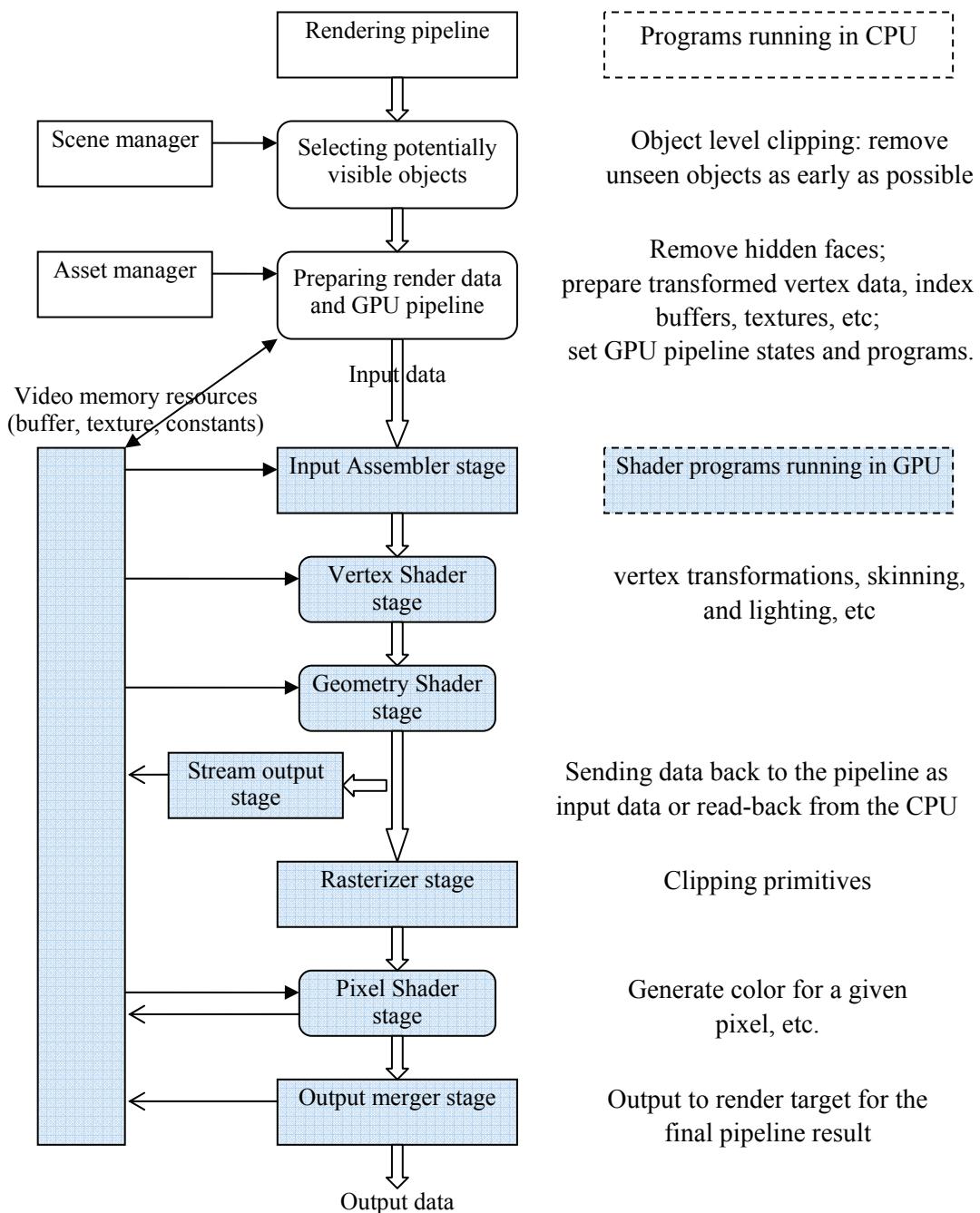


Figure 5.1 Typical rendering pipeline in 3D engine

The goal of 3D render pipeline is to make data processing parallel between CPU(s) and GPU, and process as little data as possible to render a game scene. Since we have little control of the render pipeline on the GPU, the main goal is to optimize the part on the CPU.

A good 3D rendering pipeline on the CPU side can be divided into clipping, culling, occluding and computing the right level of detail (LOD). Different game genres may emphasize certain aspects more than others to reach the desired performance. Generally, it falls into three categories: outdoors, indoors, and planetary (outer space). The first two are common. We can formally define an indoors rendering algorithm as one software piece that optimizes the rendering pipeline by quickly determining occlusions (in addition to clipping and culling) in the level geometry, thus allowing interactive rendering. This draws a distinction with outdoors rendering methods (explained in later sections), which will in turn emphasize LOD processing because view distances will be large. This characterization does not imply that indoors algorithm does not use LOD, but it is the occlusion component that will be dominant in the optimization. Regardless of the optimization goals, object level clipping always comes first. In the architecture presented in this chapter, we will only look into the object level clipping.

5.1.2 Object Level Clipping with Bounding Volumes

Object level clipping is a technique to remove an entire 3D object at an early stage of the rendering pipeline. Recall in the scene management chapter, we associated each scene objects with a view clipping object that contains pre-calculated bounding volume information. It is used in object level clipping. A pseudo code is given below. Note that we first treat the view clipping object and the camera view frustum as spheres for rough intersection test and then perform more accurate collision test using the actual shape of the two objects. After running this program, we will have a list of potentially visible objects, which will be sorted and then passed down the render pipeline for further optimization and the actual rendering.

```
Traverse quad tree scene graph with the camera frustum{
    For each sceneObject in the potentially visible terrain quad{
        clippingObject = sceneObject->GetViewCullingObject()
        if( (Test Collision (clippingObject as a 3D sphere, camera frustum as a 3D sphere) == true) and
            (Test Collision (clippingObject, camera frustum) == true) )
        {
            Add sceneObject to render object list
        }
    }
}
```

5.1.3 Object Visible Distance

The view frustum in an outdoor game may span several hundred meters. There is no need to draw everything within this region, especially when the scene contains many small objects that are barely visible in the distance. To avoid drawing small objects in the distance, we can calculate the object visible distance for different object sizes. Objects are only drawn when the object to camera distance is smaller than this value. This works even better when combined with fog effect and some alpha animation on the object. For example, when an object first appears at this distance, we play a quick animation on the object's alpha color from fully transparent to its original value and vice versa.

We can easily combine this step to the object level clipping code in the previous section. Instead of using the radius of the camera frustum for the rough collision test, we use another sphere with center at the camera origin and an object visible distance (VisibleRadius) calculated as below.

$$\text{VisibleRadius } (R) = \max((\text{fFar} - \text{Pow2}(\text{fNear} * \tan(m_fCullingAngle)) * \text{density} * (\text{fFar} - \text{fNear}) / (R * R)), \text{fNear});$$

Let R be the radius of the object. Let fNear be the near plane distance of the fog and fFar is the far plane distance of the fog. $m_fCullingAngle$ is preset to some predefined value, such as 5 degrees or $D3DX_PI/36$. We will allow an object of height $\text{fNear} * \tan(m_fCullingAngle)$ to pop out from fNear distance from the camera eye.

In the concise form, the object visible distance $v(R) = f - k / (R * R)$, where f, k are some pre-calculated values. Figure 5.2 shows the curve of $v(R)$.

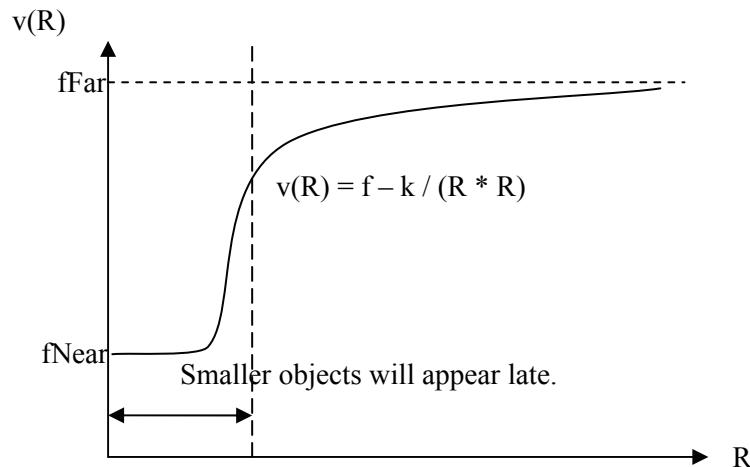


Figure 5.2 Object Level Culling: object visible distance function

The formula ensures that the pixel changes of any newly popped out object are roughly the same for objects of all sizes. I.e. small objects will only be drawn when they are very close to the camera, whereas large objects will be drawn as soon as they are in front of the far fog plane.

5.1.4 Coordinate System for Render Pipeline

One tricky thing in the render pipeline is the coordinate system in which to perform clipping, culling, occlusion testing and finally outputting to GPU. The world coordinate system seems to be the natural choice, but consider the other choices. Below is list of coordinate systems.

- **World coordinate system:** if the game designer uses meters as the world unit and that the size of game world is 40km*40km. Points in world units will fall in the range (-200000, -200000, -200000) to (200000, 200000, 200000).
- **Local coordinate system:** this is usually the coordinate system in model files. The artist uses it during 3D modeling. They are not particularly useful and are almost always transformed to its parent coordinate system.

- **Camera coordinate system:** this is the coordinate system with the origin at the current camera eye position and axis aligned with the view direction.
- **User defined coordinate system:** any other coordinate system.

In case of world coordinate system, if one uses 32 bits floating point value as vector component, there are only 4 or 5 bits left at the world boundary. For example, a player can be at position (99999.001, 0, 0); but you can not specify a player at (99999.000001,0,0) as you can with (0.000001,0,0). 0.001 meter is generally enough for moving and positioning global objects as an entity in the world. But it will not be sufficient for rendering polygon details inside the entity. So world coordinate system can not be used by the rendering pipeline unless the world size is very small and near the world origin.

A simple solution is to transform scene objects to the camera coordinate system. This effectively solves the floating point inaccuracy problems. However this is not very efficient, because the game engine must transform a scene object by a 4x4 matrix before using it in subsequent testing. A better solution is to use a user defined coordinate system, whose origin is (or is near to) the camera coordinate system's origin, and whose axis aligns with the world coordination system's axis. In ParaEngine, we call it a render coordinate system. Scene objects in world coordinate system can be transformed to this new coordinate system by simply offseting its position component. Of course, we need to transform the camera to this render coordinate system as well. By using the render coordinate system, all vertex positions processed both by CPU and GPU are near the origin and the position components of all matrices used by CPU and GPU are also near the origin. This can effectively reduce floating point inaccuracy issues during matrix and vector math.

5.1.5 Hardware Occlusion Testing

In the old days, occlusions are done mostly by software (i.e. on CPU). GPU uses Z-buffer or depth buffer for the final occlusion testing. And it is usually considered inefficient for a game engine to rely entirely on hardware (i.e. GPU) for occlusion testing.

Things have changed as GPU becomes increasingly fast. It is now even advised to throw everything to GPU, instead of wasting precious CPU time. Hardware occlusion testing is a newly supported function of GPU for doing the difficult job of occlusion testing. The idea is this: if an object contains dense polygons, yet has relatively small bounding volume, we can send the bounding volume to GPU and ask the GPU if it is visible. If the GPU says yes, we can continue sending the full polygon soup to the render pipeline; otherwise we can skip the object. In case the bounding volume is not accurate enough for occlusion testing, we can even send the complete polygon soup to GPU for occlusion testing.

In practice, we use the occlusion query results from the previous frames to decide whether to draw an object in the current frame. This is because a query is available only when the GPU have finished drawing the object. Thus, getting the query result of an object rendered in the same frame will usually cause the CPU to wait for the GPU to complete. Yet, the occlusion query results from previous frames are always available in the current frame. So, if we assume that game scenes are continuous between frames, we can use the previous occlusion query results to decide the visibility of objects in the current frame, without compromising parallelism between CPU and GPU.

This technique is very useful if mixed with other techniques in the render pipeline. For example, we can design the rendering pipeline as follows. Divide all potentially visible objects in to two lists: the first list contains solid objects that are close to the camera or having a big view angle; the second list contains transparent objects or objects that are both far away from the camera and having a small view angle. Both lists are sorted by their bounding volume's distance to the camera eye position. We then render objects in the first list from front to back with occlusion testing; and then render objects in the second list from back to front also with occlusion testing.

Figure 5.3 shows occlusion test using object's bounding volume (in this case, it is OBB). The left three objects are in the first list; the four objects on the right in blue color are in the second list. The order of drawing those objects is given by numbers.

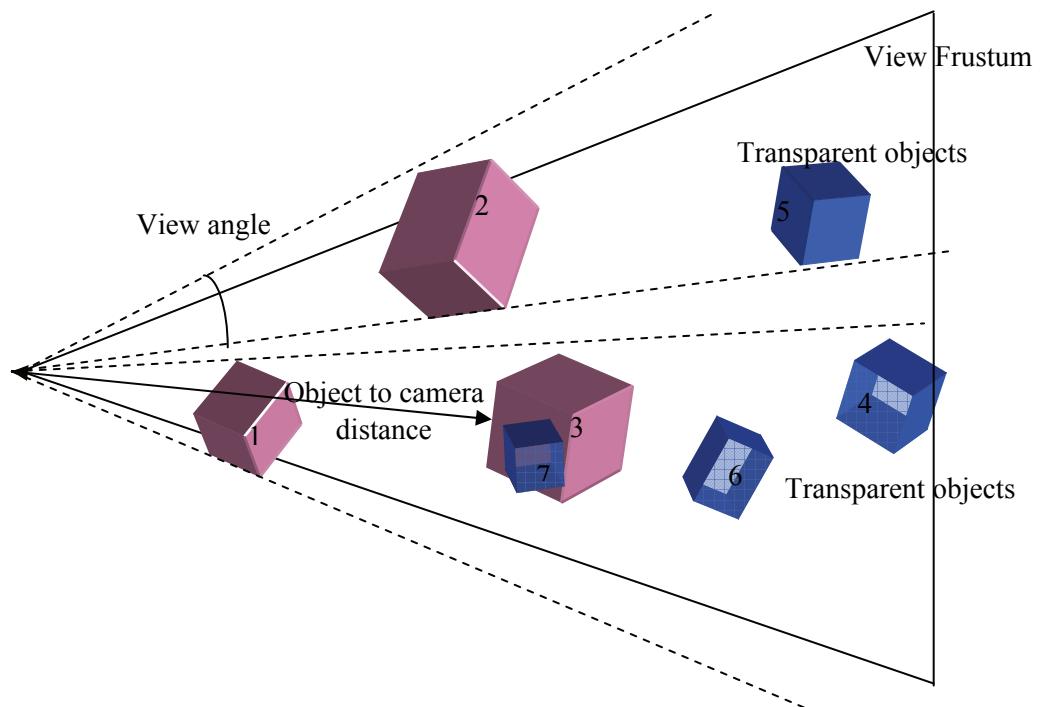


Figure 5.3 Hardware Occlusion Testing using bounding box

5.2 Building the Render Pipeline

Combining method discussed in the last section, we will now take a look at the pipeline used in ParaEngine. The pipeline is directly built into the scene manager (CSceneObject) in its member function AdvanceScene(). The following pseudo code shows the rendering pipeline.

```

Render scene (RENDER_TIMER){
    Set up scene state: camera frustum, rendering device and other user parameters, etc

    Draw the sky with depth buffer off

    Render the outdoor terrain {
        Build LOD terrain based on current camera settings.
        Draw the global terrain
    }
}

```

```

For each sceneObject in the potentially visible terrain quad{
    clippingObject = sceneObject→GetViewCullingObject()
    calculate ViewRadius of the current clipping object
    if( (Test Collision (clippingObject as a 3D sphere, sphere with ViewRadius) == true) and
        (Test Collision (clippingObject, camera frustum) == true) )
    {
        if(sceneObject is a container){
            add its child objects to the testing queue
        }
        if(sceneObject is character object){
            add it to the list_pr_Bipeds
        }
        else if(sceneObject is an ordinary mesh object){
            fObjectToCameraDist = distance from clipping object to the camera eye position
            if(sceneObject is solid object that is very near the camera or having a big view angle) {
                add pair<sceneObject, fObjectToCameraDist> to the list_pr_front_to_back
            }else{
                add pair<sceneObject, fObjectToCameraDist> to the list_pr_back_to_front
            }
        }
        else{ ... }
    }
}

Sort list_pr_front_to_back by fObjectToCameraDist
Sort list_pr_back_to_front by fObjectToCameraDist
Sort list_pr_Bipeds by asset type

for each object in list_pr_front_to_back {
    if(hardware occlusion testing with object's bounding volume is passed){
        draw object
    }
}
for each object in list_pr_back_to_front {
    if(hardware occlusion testing with object's bounding volume is passed){
        draw object
    }
}
Draw sprite objects {}

for each characterObject in list_pr_Bipeds{
    characterObject→animate();
}
Render character shadows using shadow volumes{
    ...only cast shadows on object rendered previously
}
for each characterObject in list_pr_Bipeds{
    characterObject→draw();
}

Draw missile objects { ... }
Draw the ocean { ... }
Draw particle systems in the scene {...}
Render dummy objects for debugging purposes
}

```

Generally speaking, we render indoor objects followed by outdoor objects, postponing the rendering of any small-sized but high-poly object until larger objects (occluders) have been drawn. Shadow receivers are rendered before shadow casters.

The transformation from world coordinate system to render coordinate system is performed by each view clipping object and each drawable object internally. I.e. the bounding volume returned by view clipping object is internally converted to the render coordinate system to avoid floating point inaccuracy issue.

5.3 Shader Program Management

There are two sets of rendering API supported by DirectX 9. One is for fixed function rendering pipeline (FFP), another is for programmable pipeline (PP). In version 10 of DirectX, it will only support programmable pipeline, where a game engine must supply all GPU shader programs and input streams used for different rendering stages on GPU.

However, there are some difficulties to support both rendering pipelines in a game engine. For backward compatibilities, most present day PC game engine still need to support both. In this section, we will describe one way to mix these two pipelines in a game engine. The emphasis, however, is on the programmable pipeline, or more specifically shader program management.

5.3.1 Effect File Manager

To be compatible with DirectX, our shader program is written in HLSL or High Level Shader Lanaguage. In DirectX extension library, HLSL file is also called effect files. Here, we assume that the reader understands the basics of HLSL, including shader constants, samplers, pixel and vertex shader, render passes, techniques and effect states. Such information can be found in the DirectX SDK. And we also assume that you understand the standard procedure of rendering a simple geometry using a HLSL shader program. Our focus in this section is on managing large number of shader programs and how to apply them efficiently on a very large amount of renderable scene objects.

HLSL program can be compiled into assembly code and loaded at run time. In most cases, we configure the IDE (Visual Studio) to compile HLSL in to both text and binary formats of assembly code. The text version is for human examination, the binary one is used by the game engine. We then examine the text assembly code in terms of instruction count, and we also need to take care of GPU register usage and whether the code for static and dynamic branch is being properly generated. The binary version of assembly code is what will be used by the game engine at runtime.

To compile HLSL program, one can enable custom build options in its build environment for all HLSL files and use a HLSL code compiler such as fxc.exe included in the DirectX SDK. The fxc compiler is able to generate assembly code in both text and binary format. It is common for a game engine to have 10 to 20 or even more HLSL files. They are for different scene objects and render options. See Figure 5.4. In ParaEngine, a singleton class called CEffectManager is used to manage all shader (effect) files in the game engine.

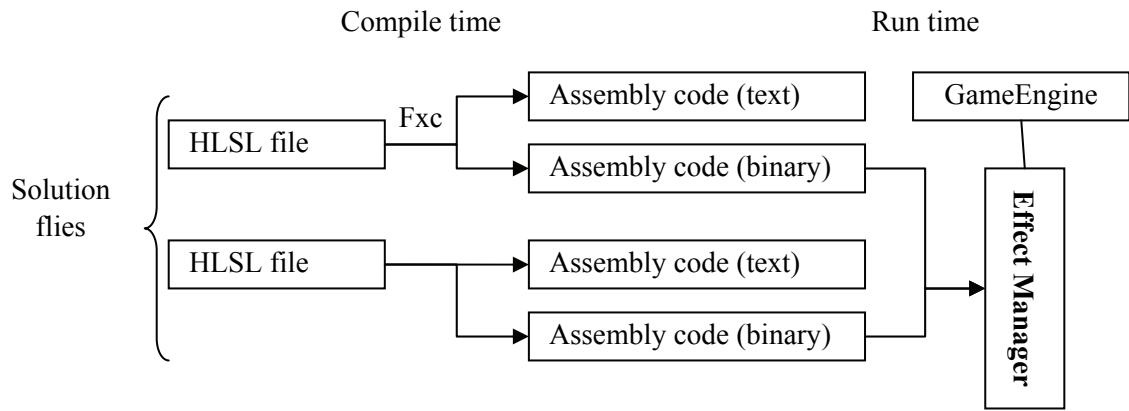


Figure 5.4 Effect Files in a Game Engine

The CEffect manager is responsible for three kinds of tasks. The first task is to dynamically load/unload effect files and switch between them. The second task is to automatically select the proper techniques for different render tasks and graphic hardware. In ParaEngine, the effect manager will default to fixed function pipeline if there are no proper techniques in the requested shader file. To be more precise, all effect files in the effect manager have an unique effect handle, where the handle 0 is reserved for the fixed function pipeline, handle 1-1024 is reserved for intenal shader file usage, handle value above 1024 is for any user defined custom shaders. The effect manager will automatically default to handle 0, if the requested effect file does not contain a valid technique. The third task of the effect manager is to expose a collection of functions for setting shared effect states, such as fog, alpha testing, alpha blending, texture samplers, and any other game engine specific parameters, etc.

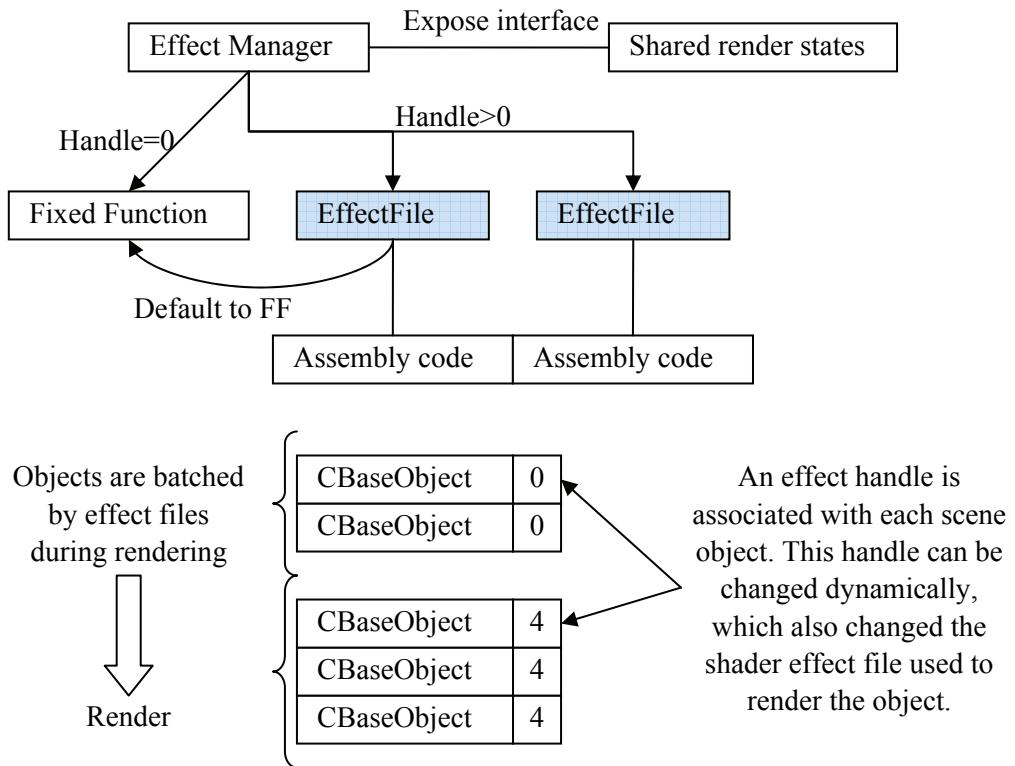


Figure 5.5 Effect handle and its association with scene objects

On the other hand, each renderable scene objects may be associated with an effect handle, which can be dynamically changed at runtime by the game logics or multistage rendering routines. Recall that in the top level render pipeline, we have batched objects in to several render groups. Now for each render group, we will further sort objects by their associated effect handles. See Figure 5.5. Switching between shader programs is one of the most time-consuming operations on GPU. Hence, in a programmable rendering pipeline, we need to batch objects by their shader programs to minimize the number of shader switch times.

5.3.2 Wrapping the Render API

The centralized effect management scheme as discussed in the previous section also provides us a good opportunity and place to wrap the DirectX render API, so that the render code elsewhere is version neutral. Wrapping the fixed function pipeline is not necessary, unless you are working on a game engine supporting both OpenGL and DirectX. Wrapping the programmable pipeline is fairly easy, we only need to wrap a couple of resource submission calls and draw calls in DirectX API; everything else, such as sampler states and shader constants, etc, are already wrapped by either the CEffectManager or CEffectFile class. Some code samples are given in the code section.

5.4 Code and Performance

5.4.1 Traversing Quad Tree

The following shows part of the rendering pipeline. It traverses the terrain quad tree to find potential visible scene objects.

```
Queue<CbaseObject*> queueNodes;
queue<CTerrainTile*> queueTiles;
CTerrainTile* pTile = (CTerrainTile*)(&m_tileRoot);
bool bQueueTilesEmpty = false;
while(bQueueTilesEmpty == false)
{
    /// add other tiles
    for(int i=0; i<MAX_NUM_SUBTILE; i++){
        if(pTile->m_subtiles[i]){
            /// rough culling algorithm using the quad tree terrain tiles test against a sphere round the eye
            if(pTile->m_subtiles[i]->TestCollisionSphere(&(vViewCenter), fViewRadius)){
                queueTiles.push( pTile->m_subtiles[i] );
            }
        }
    }
    /// go down the quad tree terrain tile to render objects
    if(queueTiles.empty()){
        bQueueTilesEmpty = true;
    }
    else
    {
        /// We will not push objects in the current terrain tile to a queue object for further view clipping.
        pTile = queueTiles.front();
        queueTiles.pop();

        /// add all solid objects to the queue for further view clipping test
        list< CBaseObject* >::iterator itCurCP, itEndCP = pTile->m_listSolidObj.end();
        for( itCurCP = pTile->m_listSolidObj.begin(); itCurCP != itEndCP; ++ itCurCP){
            if((*itCurCP)->CheckVolumnField(OBJ_VOLUMN_PERCEPTIVE_RADIUS) == false)
                queueNodes.push((*itCurCP));
        }

        if(pTile->m_listBipedVisitors.empty() == false){
            /// add all visitor biped objects to the queue.
            list< CBipedObject* >::iterator itCurCP1, itEndCP1 = pTile->m_listBipedVisitors.end();
            for( itCurCP1 = pTile->m_listBipedVisitors.begin(); itCurCP1 != itEndCP1; ++ itCurCP1){
                queueNodes.push((*itCurCP1));
            }
        }

        /// add all free space objects to the queue
        list< CBaseObject* >::iterator itCurCP, itEndCP = pTile->m_listFreespace.end();
        for( itCurCP = pTile->m_listFreespace.begin(); itCurCP != itEndCP; ++ itCurCP){
            queueNodes.push((*itCurCP));
        }
    }

    /// For any potentially visible objects in the queue,
    // perform further object-level clipping test, and draw them if the test passes.
    while(!queueNodes.empty()){
        // pop up one object
        CBaseObject* pObj = queueNodes.front();
        CBaseObject* pViewClippingObject = pObj->GetViewClippingObject();
        queueNodes.pop();
    }
}
```

```

ObjectType oType = pObj->GetMyType();
bool bDrawObj, bDrawChildren;

{
    ...// object level clipping code here
    ...// set bDrawChildren
}

/// push its child objects to the queue
if(bDrawChildren){
    list< CBaseObject* >::iterator itCurCP, itEndCP = pObj->GetChildren()->end();
    for( itCurCP = pObj->GetChildren()->begin(); itCurCP != itEndCP; ++ itCurCP){
        queueNodes.push(* itCurCP);
    }
}
} //while(!queueNodes.empty())

}//while(!queueTiles.empty())
... // lots of code omitted

```

5.4.2 Render Coordinate System Transformation

The following code shows how render coordinate system is used during object level clipping and object rendering.

```

D3DXVECTOR3 CSceneObject::GetRenderOrigin(){
    return m_vRenderOrigin;
}

D3DXVECTOR3 CBaseObject::GetWorldPosition(){
    D3DXVECTOR3 vPos;
    GetPosition(&vPos);
    return vPos;
}

D3DXVECTOR3 CBaseObject::GetRenderOffset(){
    D3DXVECTOR3 vPos;
    GetPosition(&vPos);
    return (vPos-CGlobals::GetScene()->GetRenderOrigin());
}

void CViewCullingObject::GetRenderVertices(D3DXVECTOR3 * pVertices, int* nNumber)
{
    int nNum = 8;
    *nNumber = nNum;
    pVertices[0].x = m_vMin.x; pVertices[0].y = m_vMin.y; pVertices[0].z = m_vMin.z;
    ... // code setting pVertices[1] to pVertices[6] omitted
    pVertices[7].x = m_vMax.x; pVertices[7].y = m_vMax.y; pVertices[7].z = m_vMin.z;

    D3DXMATRIX mat = m_mWorldTransform;
    /** this will make the components of world transform matrix consistent (i.e. of the same magnitude.), hence it will correct floating point calculation imprecision.*/
    D3DXVECTOR3 vOrig = CGlobals::GetScene()->GetRenderOrigin();
    mat._41 -= vOrig.x;
    mat._42 -= vOrig.y;
    mat._43 -= vOrig.z;

    for(int i=0; i<nNum;i++){
        D3DXVec3TransformCoord( &pVertices[i], &pVertices[i], &mat);
    }
}

```

```

    }

HRESULT CMeshObject::Draw( SceneState * sceneState){
    ... // code omitted here
    // using the render coordinate system to render mesh
    D3DXVECTOR3 vPos = GetRenderOffset();
    D3DXMatrixTranslation( & mx, vPos.x, vPos.y, vPos.z);
    mxWorld = m_mxLocalTransform* mx;

    pd3dDevice->SetTransform( D3DTS_WORLD, & mxWorld );
    ... // draw mesh here
}

```

5.4.3 Hardware Occlusion Testing

The code shown here is written with DirectX 9.0C using fixed function pipeline.

```

...// the following code is executed for every object in the post rendering list.
if(m_bEnableOcclusionQuery && d3dQuery){
    /** Occlusion culling algorithm is used after drawing all solid objects. */
    CBaseObject* pViewClippingObject = (*itCurCP).m_pRenderObject->GetViewClippingObject();

    // Start the query
    pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    d3dQuery->Issue( D3DISSUE_BEGIN );
    // Make sure that no pixels get drawn to the frame buffer
    pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_ZERO );
    pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_ONE );
    // Render the occlusion object: in this case, it is the bounding box of the clipping object
    pViewClippingObject->DrawOcclusionObject(&sceneState);
    pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
    pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
    pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
    // End the query, get the data
    d3dQuery->Issue( D3DISSUE_END );

    pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
    // Loop until the data becomes available
    DWORD pixelsVisible = 0;
    while (d3dQuery->GetData((void *) &pixelsVisible,
        sizeof(DWORD), D3DGETDATA_FLUSH) == S_FALSE)
    {

        if( pixelsVisible > 0 ){
            // render the object.
            (*itCurCP).m_pRenderObject->Draw(&sceneState);
            ++ itCurCP;
        }else{
            // if not visible, remove from the list.
            itCurCP = sceneState.listPRTTransparentObject.erase(itCurCP);
            nOccludedObjCount++;
        }
    }
}

```

5.4.4 Effect Management

Header code of effect manager.

```

/** manager all effects file used by the game engine.*/
class EffectManager: public AssetManager <CEffectFile>
{
public:
    EffectManager();
    ~EffectManager();

    CEffectFile* GetEffectByHandle(int nHandle);

    CEffectFile* MapHandleToEffect(int nHandle, CEffectFile* pNewEffect);

    virtual bool DeleteEntity(AssetEntity* entity);
    virtual void InitDeviceObjects();
    virtual void DeleteDeviceObjects();
    virtual void RestoreDeviceObjects();
    virtual void InvalidateDeviceObjects();
    void Cleanup();

    /** get the shadow map object. */
    CShadowMap* GetShadowMap();
    /** get the glow effect object. */
    CGlowEffect* GetGlowEffect();

    /** Start effect by handle. It set the proper technique of the effect.
     * the vertex declaration will also be set or created.
     * @param nHandle: the effect handle.
     * @param pOutEffect: pointer to retrieve the effect object which is currently selected.
     * @return: return true if succeeded.
     */
    bool BeginEffect(int nHandle, CEffectFile** pOutEffect = NULL);
    /** end the effect, this function actually does nothing. */
    void EndEffect();

    /** predefined vertex declaration. */
    enum VERTEX_DECLARATION
    {
        S0_POS_TEX0, // all data in stream 0: position and tex0
        S0_POS_NORM_TEX0, // all data in stream 0: position, normal and tex0
        S0_S1_S2_OCEAN_FFT, // for FFT ocean
        S0_POS_NORM_TEX0_TEX1, // all data in stream 0: position, normal tex0 and tex1
        MAX_DECLARATIONS_NUM,
    };

    /** Get declaration by id
     * @param nIndex: value is in @see VERTEX_DECLARATION
     * @return: may return NULL.*/
    LPDIRECT3DVERTEXDECLARATION9 GetVertexDeclaration(int nIndex);
    /** Set declaration by id
     * @param nIndex: value is in @see VERTEX_DECLARATION
     * @return: return S_OK if successful. */
    HRESULT SetVertexDeclaration(int nIndex);

    /** load the default handle to effect file mapping.
     * @param nLevel: the higher this value, the more sophisticated shader will be used.
     * The default value is 0, which is the fixed programming pipeline without shaders.
     * 10, VS PS shader version 1
     * 20, VS PS shader version 2
     * 30, VS PS shader version 3
     * please note that any user defined mapping will be cleared.
    */
}

```

```

*/
void LoadDefaultEffectMapping(int nLevel);
enum EffectTechniques
{
    /// normal rendering
    EFFECT_DEFAULT = 0,
    /// shadow map generation technique
    EFFECT_GEN_SHADOWMAP,
    /// rendering the model with shadow map
    EFFECT_RENDER_WITH_SHADOWMAP,
    /// fixed function pipeline
    EFFECT_FIXED_FUNCTION,
};

/** set all effect files to a specified technique.If the effect does not have the specified technique
nothing will be changed.
@param nTech: the technique handle. */
void SetAllEffectsTechnique(EffectTechniques nTech);
/** current technique in the effect file*/
EffectTechniques GetCurrentEffectTechniqueType();

int GetCurrentTechHandle();
CEffectFile* GetCurrentEffectFile();

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

// The following functions set or retrieve global effect states which are shared by all effect files.  

// They are designed to look like the fixed pipeline programming interface of DirectX9  

//  

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

HRESULT SetMaterial(D3DMATERIAL9 *pMaterial);
HRESULT SetLight(DWORD Index, CONST D3DLIGHT9 *pLight);
HRESULT LightEnable(DWORD Index, BOOL Enable);
HRESULT SetRenderState(D3DRENDERSTATETYPE State, DWORD Value);
HRESULT SetTexture(DWORD Stage, LPDIRECT3DBASETEXTURE9 pTexture);
HRESULT SetTransform(D3DTRANSFORMSTATETYPE State, CONST D3DMATRIX *pMatrix);
HRESULT GetTransform(D3DTRANSFORMSTATETYPE State,D3DMATRIX * pMatrix);

/** Get the current world transformation matrix which is used by the effect.
@see GetTransform()*/
D3DXMATRIX& GetWorldTransform();
D3DXMATRIX& GetViewTransform();
D3DXMATRIX& GetProjTransform();

/** update the transformation for both the fixed and programmable pipeline.
@param pWorld: world transformation, only set if it is true,
@param pView: camera view transformation, only set if it is true,
@param pProjection: camera projection transformation, only set if it is true, */
void UpdateD3DPipelineTransform(bool pWorld, bool pView,bool pProjection);

/** enable or disable fog.*/
void EnableFog(bool bEnabled);

/** clip plane state */
enum ClipPlaneState{
    ClipPlane_Disabled,
    ClipPlane_Enabled_WorldSpace,
    ClipPlane_Enabled_ClipSpace,
};

}

```

```

void EnableClipPlane(bool bEnable);

void SetClipPlane(DWORD Index, const float * pPlane, bool bClipSpace);

// ... many game engine specific render state function ignored here

private:
    /** whether use lighting: global sun light and local point lights */
    bool m_bEnableLocalLighting;
    /** whether use fog*/
    bool m_bUseFog;
    /** if true, setting the alpha testing parameter of the effect files will have no effect. */
    bool m_bDisableD3DAlphaTesting;
    /** whether using shadow map */
    bool m_bUsingShadowMap;
    /** whether full screen glow effect is used. */
    bool m_bIsUsingFullScreenGlow;
    /** glowness controls the intensity of the glow in the full screen glow effect.
        0 means no glow, 1 is normal glow, 3 is three times glow. default value is 1. */
    D3DXCOLOR m_colorGlowness;
    /** the glow technique to use. */
    int m_nGlowTechnique;

    /** if true, setting the culling mode of the effect files will have no effect. */
    bool m_bDisableD3DCulling;

    ClipPlaneState m_ClipPlaneState;
    bool m_bClipPlaneEnabled;
    static const int MaxClipPlanesNum = 3;
    D3DXPLANE m_ClipPlaneWorldSpace[MaxClipPlanesNum];
    D3DXPLANE m_ClipPlaneClipSpace[MaxClipPlanesNum];

    CShadowMap* m_pShadowMap;
    CGlowEffect* m_pGlowEffect;

    /** all vertex declarations*/
    LPDIRECT3DVERTEXDECLARATION9 m_pVertexDeclarations[MAX_DECLARATIONS_NUM];
// ... several game engine specific class members ignored here
};

```

5.4.5 Performance Discussion

The rendering pipeline presented in this chapter is effective for both outdoor and indoor game scenes. In a hybrid game engine, the scene objects may be ordinary mesh objects, such as trees, stones, tables, bottles, etc; or characters and creatures with dense polygons; or large meshes with indoor rooms, such as camps, buildings and castles, etc. When rendering large mesh objects, complex hidden surface removal algorithm is usually applied in order to send fewer polygons to GPU. Popular algorithms are BSP, Octree, Portal Rendering, etc. However, as GPU is becoming increasingly powerful, it is now possible to send an entire moderate-sized mesh to GPU for occlusion testing and rendering. This not only saves CPU times, but also makes level editing easier, because it has little constraints and no human intervention when making 3D models or designing game levels.

To test the performance, we have performed a test as follows.

- Select a hybrid game scene with both indoor and outdoor objects
- (1) Render the scene using solely object level clipping and hardware occlusion testing
- (2) Render the scene using BSP based software occlusion testing for indoor objects.

Method (2) does not exhibit higher performance than (1) with NVIDIA 6800 GPU. Figure 5.6 shows the game scene we have chosen.



Figure 5.6 Hardware Occlusion Testing using bounding box

5.5 Summary and outlook

Rendering pipeline is the most important and distinguishing component of a 3D engine. The design of a rendering pipeline depends on game requirement and hardware architecture. It is common for engine programmers to rewrite the rendering pipeline several times, during the development of a game engine or a specific game title.

We think that there is no official version of a general rendering pipeline. Engine programmers have the freedom to upgrade it as new graphics functions are added. However, in the design of a rendering pipeline, one should avoid intertwining it with other game engine modules, such as AI, physics and scripting.

Computer graphics hardware evolves very fast and it causes the software architecture of a game engine to evolve as well. For example, the evolutionary BSP based scene management is losing popularity with modern hardware; whereas Octree and Portal-based rendering are gaining popularity. On the other hand, GPU memory is getting cheaper; 2GB video memory will soon become common. Multi-core computer is getting in, the new Xbox 360 already have two CPUs. Parallel programming may soon become common in next generation game engine. Microsoft supports OpenMP (a parallel programming) extension to C++ in visual studio 2005. Although this will not bring radical changes to the architecture of 3D engine, but it shows how a computer game engine evolves with incremental advancement in computer hardware.

Techniques we used today may be outdated at the time they are used for the actual game. So, agree with us or not, unless you are programming for a specific game title or doing specific research, we do not advise you to spend too much time on eye-catching graphics effects. This will result in lots of useless code in your rendering pipeline and many unused peripheral tools created for the modelers. In case you are extending your 3D engine, always foresee as much as possible. For example, you should be able to predictive the mainstream GPU and CPU specifications in the next 1 or 2 years and stick to it when extending your engine.

Chapter 6 Special Scene Objects

This chapter covers the design and implementation of several special scene objects which are too trivial to be put in separate chapters. This chapter is written for programmers mostly.

6.1 Sky

In a game engine, the sky background is rendered as an ordinary mesh object with Z-buffer disabled and origin fixed at the current camera eye position. The sky mesh object is usually a box, a dome or a plane, depending on the game requirement. The size of the sky mesh is arbitrary (unit size is fine), but since it moves with the camera, it will give the illusion of a sky background in the infinite distance. For more information about the sky effect please read the Fog and Sky section in [Chapter 11](#).

6.2 Sun and Sun Light

The sun is modeled by a special global object called (CSunLight). It contains parameters for the sun color, directions, and light scattering data, time of day, etc. The sun direction is related to the time of day. Other scene objects can query the sun object for its current parameters. For example, we can take the sun parameter in to consideration when rendering the fog; and take the sun direction in to consideration when rendering shadows, and takes the scattering data in to consideration when rendering ocean reflections and refractions. If the game engine has a simulated weather system, it may include the sun object as its member. The following is the header file for the sun object.

```
class CLightScatteringData{
private:
    float m_henyeyG;
    float m_rayleighBetaMultiplier;
    float m_mieBetaMultiplier;
    float m_inscatteringMultiplier;
    float m_extinctionMultiplier;
    float m_reflectivePower;
    sLightScatteringShaderParams m_shaderParams;
public:
    ...// functions omitted
};

/**
 * Modeling the global sun and its directional light
 */
class CSunLight
{
public:
    CSunLight();
    ~CSunLight();
private:
    D3DLIGHT9 m_light;
    float    m_seconds; // how many seconds passed since the last record
    float    m_fYaxisRotation;
    float    m_fMaxAngle;
    float    m_fCurrentAngle; // in the range [-m_fMaxAngle, +m_fMaxAngle]
```

```

float m_fDayLength;
D3DXCOLOR m_colorSun; // color of the sun
CLightScatteringData m_LightScatteringData;
private:
    void RecalculateLightDirection();
public:
    D3DLIGHT9* GetD3DLight(){return &m_light;}
    /// set time of day in seconds
    void SetTimeOfDay(float time){m_seconds = time;}
    /// get time of day in seconds
    float GetTimeOfDay(){return m_seconds;}
    /// set position of the time
    void SetPosition(const D3DXVECTOR3& pos){m_light.Position = pos;}
    /// advance time is in seconds
    float AdvanceTimeOfDay(float timeDelta);
    /// get the sun color
    D3DXCOLOR GetSunColor(){return m_colorSun;}
    /// get the sun direction vector
    D3DXVECTOR3 GetSunDirection(){return m_light.Direction;}
    /// get the sun Ambient Hue
    D3DXCOLOR GetSunAmbientHue(){return D3DXCOLOR(m_light.Ambient);}
    /// get light Scattering Data object
    CLightScatteringData* GetLightScatteringData(){return &m_LightScatteringData;}
    ..// functions omitted
};

```

6.3 Particle System

A particle system is usually implemented as a group of animated sprites which are always facing the camera. They are a cheap way to achieve seemingly complex graphics effect, such as smokes, fires, magic, etc.

6.3.1 Defining a Particle System

An instance of particle system is a list of particles, each of which is represented by current position, speed, acceleration (gravity) direction, origin, size, life, maximum lifetime, texture tile index, and color. See the specifications in C++ structures as follows.

```

Struct Particle {
    Vec3D pos, speed, down, origin;
    float size, life, maxlife;
    int tile;
    Vec4D color;
};

/** it represents the instance of the particle system.*/
struct ParticleList{
    std::list<Particle> particles;
    /** whether to use absolute world coordinate system, so that when particles leave its source, it will be
    an independent object in the world coordinate system.*/
    bool m_bUseAbsCord:1;
    /** the last render origin when the update() function is called.*/
    Vec3D m_vLastRenderOrigin;
    /** whether this object is updated in the current frame.*/
    bool m_bUpdated:1;
    /** whether this object should be rendered (in camera frustum)*/
    bool m_bRender:1;
    /** current animation */
}

```

```

int m_anim;
/** current frame */
int m_time;
/** remaining time for spawning new particles */
float m_rem;
public:
    ParticleList():m_bUseAbsCord(true), m_vLastRenderOrigin(0,0,0),m_bUpdated(false),m_anim(0),
    m_time(0), m_rem(0){};
};

```

The same particle system can have many particle system instances in the scene. A particle system animates particles in all of its particle system instances according to predefined animation data and other factors such as gravity. Usually we need a particle editor for artists to make animations of a particle system. The following things can be animated in a standard particle system: speed, variation, spread, gravity, lifespan, rate, size, color etc. The follow code shows a standard particle system data structure.

```

class ParticleSystem {
    Animated<float> speed, variation, spread, lat, gravity, lifespan, rate, areal, areaw, grav2;
    Vec4D colors[3];
    float sizes[3];
    float mid, slowdown, rotation;
    Vec3D pos;
    TextureEntity* texture;
    ParticleEmitter *emitter;
    int blend,order,type;
    int rows, cols;
    std::vector<TexCoordSet> tiles;
    void initTile(Vec2D *tc, int num);
    bool billboard:1;

    Bone* parent; // to which it is attached in a 3D model
public:
    CParaXModel *model;
    float tofs;
    /** instances of the particle systems. mapping from the owner object, typically this is a scene object,
    to the particle list of that owner */
    map <void*, ParticleList*> m_instances;

    void init(CParaFile &f, ModelParticleEmitterDef &mta, int *globals);

    void update(float dt);
    /**
     * @param dt: time delta
     * @param vOffset: all position and origin will be offset by this value during animation.
     *      this is usually the render origin offset between two consecutive calls.
     * @param instancePS: particle system instance.
     * @return: return true if there is still particles in the instance.
     */
    bool AnimateExistingParticles(float dt, const D3DXVECTOR3& vOffset, ParticleList* instancePS);

    /** get the current particle system instance according to the current scene object.
     * @param bForceCreate: if this is true, the instance will be created if not found.
     */
    ParticleList* GetCurrentInstance(bool bForceCreate=true);
    void setup(int anim, int time);
    /** draw the current instance, if it is a non-absolute particle system instance. */
    void draw();
    /** draw a specified instance.*/
}

```

```

void drawInstance(ParticleList* instancePS);
/** draw all absolute instances.This is called for batch-rendering global(absolute) particle instances.*/
void drawAllAbsInstances();
...// some functions and members omitted
};

```

The particle system class keeps a list of all of particle system instances. A particle system instance is always associated with an owner scene object, such as a missile, a mesh, or a character object. The instance may be global or local. Global particles remain in the scene even after its owner object is destroyed and they are rendered in the global world coordinate system. The scene manager keeps a reference to all active particle systems and their instances. It will automatically delete instances that are no longer active in the current frame. It will also animate and render these global instances. Particles in global particle systems should have a short (finite) life time, in order to clean themselves quickly after its owner is released.

6.3.2 Animating Particles

The particle engine mathematics that a particle system uses is very simple. It is a two-step process. During each frame move, it first animates existing particles in a particle system instance, removing particles that are out of its life time; secondly it generates new particles, giving them initial position, speed, etc. The module to spawn new particles is usually called particle emitters. We can use different emitters to spawn particles in different patterns.

Animating an existing particle by a delta time (Δt) is this:

Define a function called keyframes as below:

`keyframes(type, animation_id, frame_number)` = retrieves a key frame based animation data of the given type at the given time and with given animation id.

Parameter description:

- time: current frame time of the particle
- animID: the animation ID of the particle
- Δt : time delta
- a: acceleration of the particle.
- s: current position of the particle
- v: speed of the particle
- life: current life time of the particle
- size: size of the particle
- color: particle color
- SLOWDOWN, MAXLIFE, MID, SIZE0, SIZE1, SIZE2, COLOR0, COLOR1, COLOR2: constants

Animating particles mathematically:

$$\vec{a} = \text{keyframes}(\text{type}(\vec{a}), \text{animID}, \text{time})$$

$$\vec{s} = \vec{s} + (\vec{v} + (\vec{a} \times \Delta t)) \times \Delta t, \text{ if without slowing down factor}$$

$$\vec{s} + (\vec{v} + (\vec{a} \times \Delta t)) \times e^{(-SLOWDOWN \times life)} \times \Delta t, \text{ if with slowing down factor}$$

$$life = life + \Delta t$$

Define interpolation function $f(x, mid, a, b, c) =$

$$(x/mid) \times a + (1-x/mid) \times b, \text{ if } x \leq mid$$

$$(x-mid)/(1-mid) \times b + (1-(x-mid)/(1-mid)) \times c, \text{ if } x > mid$$

size = $f(life/MAXLIFE, MID, SIZE0, SIZE1, SIZE2)$

color = $f(life/MAXLIFE, MID, COLOR0, COLOR1, COLOR2)$

The following shows the animation code of a particle system.

```
bool ParticleSystem::AnimateExistingParticles(float dt, const D3DXVECTOR3& vOffset, ParticleList* instancePS){
    if(instancePS==NULL)
        return false;
    std::list<Particle> & particles = instancePS->particles;

    int manim = instancePS->m_anim;
    int mtime = instancePS->m_time;
    float grav = gravity.getValue(manim, mtime);
    float mspeed = 1.0f;

    std::list<Particle>::iterator it;
    for (it = particles.begin(); it != particles.end(); ) {
        Particle &p = *it;
        p.speed += p.down * grav * dt;

        if (slowdown>0) {
            mspeed = expf(-1.0f * slowdown * p.life);
        }
        p.pos += p.speed * mspeed * dt;
        if(instancePS->m_bUseAbsCord)
        {
            p.pos += (const Vec3D&)vOffset;
            p.origin += (const Vec3D&)vOffset;
        }

        p.life += dt;
        float rlife = p.life / p.maxlife;
        // calculate size and color based on lifetime
        p.size = lifeRamp<float>(rlife, mid, sizes[0], sizes[1], sizes[2]);
        p.color = lifeRamp<Vec4D>(rlife, mid, colors[0], colors[1], colors[2]);

        // kill off old particles
        if (rlife >= 1.0f)
            it = particles.erase(it);
        else
            ++it;
    }
    return (particles.size() > 0);
}

void ParticleSystem::update(float dt){
    /* get the particle system instance for the current scene object.*/
    ParticleList* instancePS= GetCurrentInstance();
```

```

if(instancePS == NULL)
    return;
/// add this particle system to the scene state. so that the scene state will automatically
/// maintain the life of this particle system instance.
CGlobals::GetSceneState()->AddParticleSystem(this);
// mark as updated instance
instancePS->m_bUpdated = true;

std::list<Particle> & particles = instancePS->particles;

Vec3D vRenderOriginOffset;
D3DXMATRIX mWorld;
Vec3D vRenderOrigin;

if(instancePS->m_bUseAbsCord){
    CGlobals::GetRenderDevice()->GetTransform(D3DTS_WORLD, &mWorld);
    vRenderOrigin = *((Vec3D*)&(CGlobals::GetScene()->GetRenderOrigin())));

    vRenderOriginOffset = instancePS->m_vLastRenderOrigin - vRenderOrigin;
    instancePS->m_vLastRenderOrigin = vRenderOrigin;// update render origin
}

/** animate existing particles.*/
AnimateExistingParticles(dt, (const D3DXVECTOR3&)vRenderOriginOffset, instancePS);

/** spawn new particles */
if (emitter) {
    int manim = instancePS->m_anim;
    int mtime = instancePS->m_time;
    float frate = rate.getValue(manim, mtime);
    float ftospawn = (dt * frate / flife) + instancePS->m_rem;
    if (ftospawn < 1.0f) {
        instancePS->m_rem = ftospawn;
        if (instancePS->m_rem<0) instancePS->m_rem = 0;
    }
    else {
        int tospawn = (int)ftospawn;
        instancePS->m_rem = ftospawn - (float)tospawn;
        for (int i=0; i<tospawn; i++) {
            Particle p = emitter->newParticle(manim, mtime);
            if(instancePS->m_bUseAbsCord){

// transform to world coordinate system, suppose that the world transform is in device already set
                D3DXVECTOR4 tmp;
                D3DXVec3Transform(&tmp, &p.pos, &mWorld);
                p.pos.x = tmp.x;
                p.pos.y = tmp.y;
                p.pos.z = tmp.z;

                D3DXVec3Transform(&tmp, &p.origin, &mWorld);
                p.origin.x = tmp.x;
                p.origin.y = tmp.y;
                p.origin.z = tmp.z;
            }
            // sanity check:
            if (particles.size() < MAX_PARTICLES) particles.push_back(p);
        }
    }
}
}

```

Many kinds of emitters can be used to spawn new particles for a particle system. The following shows a simple circle emitter that randomly emits particles from an origin in all directions.

```
Particle CircleEmitter::newParticle(int anim, int time){
    Particle p;
    float l = sys->areal.getValue(anim, time);
    float w = sys->areaw.getValue(anim, time);
    float spd = sys->speed.getValue(anim, time);
    float var = sys->variation.getValue(anim, time);

    float t = randfloat(0,2*PI);
    Vec3D bdir(0, l*cosf(t), w*sinf(t));
    p.pos = sys->pos + bdir;
    p.pos = sys->parent->mat * p.pos;
    if (bdir.lengthSquared() == 0) p.speed = Vec3D(0,0,0);
    else {
        Vec3D dir = sys->parent->mrot * (bdir.normalize());
        p.speed = dir.normalize() * spd * (1.0f+randfloat(-var,var));
    }
    p.down = sys->parent->mrot * Vec3D(0,-1.0f,0);
    p.life = 0;
    p.maxlife = sys->lifespan.getValue(anim, time);
    p.origin = p.pos;
    p.tile = randint(0, sys->rows*sys->cols-1);
    return p;
}
```

6.3.3 Drawing Particles

Many scene objects such as mesh and creatures may contain particles system. However, the drawing of particle systems is usually postponed until most other scene objects have been drawn. The scene manager maintains a list of active particle systems in the scene and the rendering pipeline will draw them in a batch. When scene objects containing particle systems are rendered in the rendering pipeline, their associated particle systems will not be immediately drawn, instead they will be put to the particle system list maintained by the scene manager. Recall the rendering pipeline in Chapter 5. We render all particle systems in the scene last in a batch.

Each particle is rendered as a sprite facing the camera. There are many ways to do so. Some GPU supports point sprite, so that one can send particles to GPU as points. Alternatively, you can compute the object facing by CPU and render particles as quads (two triangles) in the traditional way. This will give you more freedom on the way a particle is rendered. Perhaps, writing your own shaders for particle rendering is a better choice which combines the speed of GPU sprite method and the flexibility of the CPU quad method. One can accelerate particle rendering by using the face normal of the gravity point for all particles which in a particle system instance. Figure 6.1 shows the idea and Figure 6.2 is our rendering result using this approximated method.

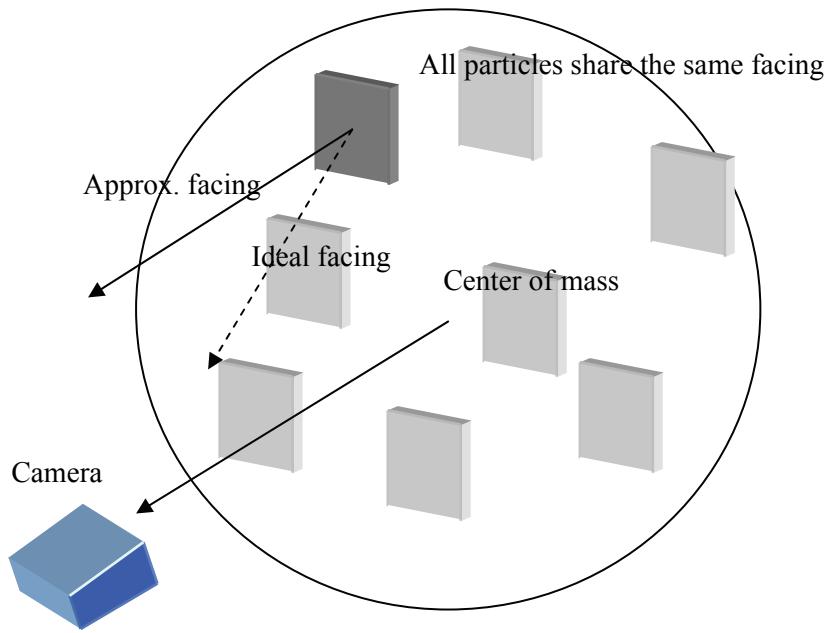


Figure 6.1 Particle rendering with approximated facing using center of mass



Figure 6.2 Particle Systems in Games

6.4 Weather System

We will end this chapter by putting functionalities introduced in this chapter together to a game module called weather system. Figure 6.3 shows components of a weather system and game modules which are related to the weather in the game world.

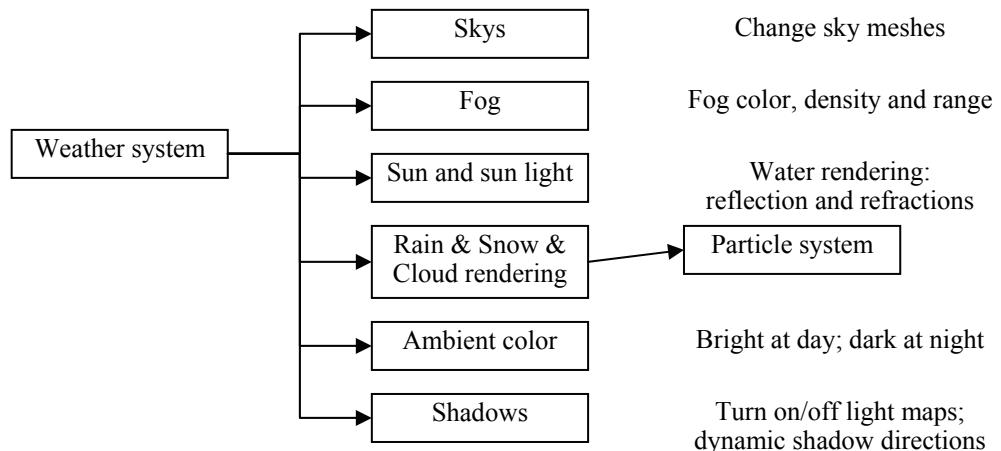


Figure 6.3 Weather system

Chapter 7 Picking

Picking is about casting a 3D ray (**origin**, **direction**, length) and finding objects which intersect with the ray. Information such as the intersection point and intersected face normal can be returned on demand. This chapter focuses on the implementation and different uses of picking in a game engine. The most common use is to find the scene object that intersects with the user's mouse cursor. In a 3D game, the screen position (x,y) of the mouse cursor is first converted to a 3D ray using the current camera and projection settings; then this 3D ray will be used to pick out the nearest scene object that intersects with it.

7.1 Foundations

7.1.1 Background

In the old days, ray picking is a fundamental component of a game engine. It provides a ray based sensor for querying the game world. A number of game engine modules, such as physics, camera, and AI, depend on picking functions to get information about the game world. For example, the physic module (imaging a flying missile) use ray based sensors to detect obstacles ahead and terrain height positions beneath. The camera uses ray based sensors to detect if it is very close to a physical surface and should move way from it. An intelligent character uses a group of sensor rays to deduct terrain and walls in its vicinity. Of course the most common application is mouse picking on 3D scene objects.

The data structure we discussed for storing scene objects (see **Chapter 4**) are also optimized for fast ray picking. For example, BSP tree is ideal for accuracy ray picking at triangle level. Quad-tree and Octree can also perform ray picking very fast at the object level.

As technology evolves, we have a few other options to “touch” objects in the game world, such as using a bounding sphere or box instead of a ray. Moreover, this group of collision detection functionalities has been elevated to a new level of game physics, which involves rigid body dynamics simulation and collision detection of any shapes. As we will see in later chapters, game physics nowadays are implemented as middleware or an independent module, which usually has its own internal data presentation. However, sometimes, we can not rely solely on a separate game physics module for the following reasons:

- Data duplications: we will have two duplicate copies of game data (such as a mesh) in two file formats: one for physics and one for rendering. For example, we do not like the global terrain mesh to be duplicated in a physics simulation middleware.
- High level collision detection: sometimes we do not need collision information at the triangle level. Instead, we only need to perform collision detection in object level. For example, mouse ray picking is usually an object level task.
- Special game requirement: a general physics middleware can not provide everything we need for a specific game title. For example, we may want some mesh to change shapes (morphing) or dig holes on mesh surfaces, etc, while still providing real time collision detection for them. Such things can not be effectively achieved through a unified data structure provided by the physics middleware.

To sum up, a game engine usually needs to combine several different implementations to achieve the required collision detection tasks in a diverse game world. In ParaEngine, the global terrain, the ocean manager, the physics engines and the scene picking function all provide similar collision detection functions for their governed objects. Scene picking function is usually an implementation for ray based collision detection at object level. We will cover this particular one in this chapter.

7.1.2 Picking Mathematics

The mathematics used in ray picking involves collision detection between a ray and an oriented 3D box. Some math library usually includes collision detections between the following basic shapes: ray (and line segment), triangle mesh, sphere, axis aligned bounding box (AABB), oriented bounding box (OBB). We advise novice engine programmers to study some open source implementation, such as the ODE physics engine. For object level ray picking, we will only need collision detection between a ray and an axis aligned bounding box. Optionally, we can have a collision detection function between a ray and an oriented bounding box.

The following lists the mathematical presentations of the ray, AABB and OBB.

- A ray is a half-line $P(t) = m\text{Orig} + m\text{Dir} * t$, with $0 \leq t \leq +\infty$
- An AABB is box which is aligned to x,y,z axis. It can be defined by two points ($v\text{Min}$, $v\text{Max}$), where $v\text{Min}.x \leq v\text{Max}.x$, $v\text{Min}.y \leq v\text{Max}.y$, $v\text{Min}.z \leq v\text{Max}.z$. It may also be defined by the box's center $v\text{Center}$ and its extents ($v\text{Extents}$) from its center along the x, y, z axis.
- An OBB is a box which is arbitrarily oriented in space. It can be defined as an AABB plus a rotation matrix ($m\text{Rot}$).

7.1.2.1 Ray-AABB Collision Detection

We will study a fast algorithm for ray and AABB overlapping test without returning the intersection point or ray-box distance. The algorithm given below will return true if the ray intersects the box, and false otherwise.

Assume ray ($m\text{Orig}$, $m\text{Dir}$) and AABB (center, extents). Let $D = m\text{Orig}-\text{center}$. Project the ray and the AABB to x,y,z planes respectively, we will get a 2D ray and a rectangle centered at the origin during each of the three projections. See Figure 7.1. Perform the following tests.

- Test 1: Consider a box with the same center as the AABB but with extents $|D|$. The ray origin is on the surface of the new box. If the ray points out of the new box and that the ray origin is outside AABB, the algorithm terminates and return false. See Figure 7.1. This test will quickly exclude outward pointing rays like Ray 1 (red) in the figure.
- Test 2: Consider the three projections on the x,y,z planes. If in any projection, the ray is outside the angle range denoted by the grey arc in the figure, the algorithm terminates and return false. This test will further remove rays like Ray 2 (blue) in the figure.
- If none of the above test returns, it means that the ray intersects with the box and the algorithm returns true.

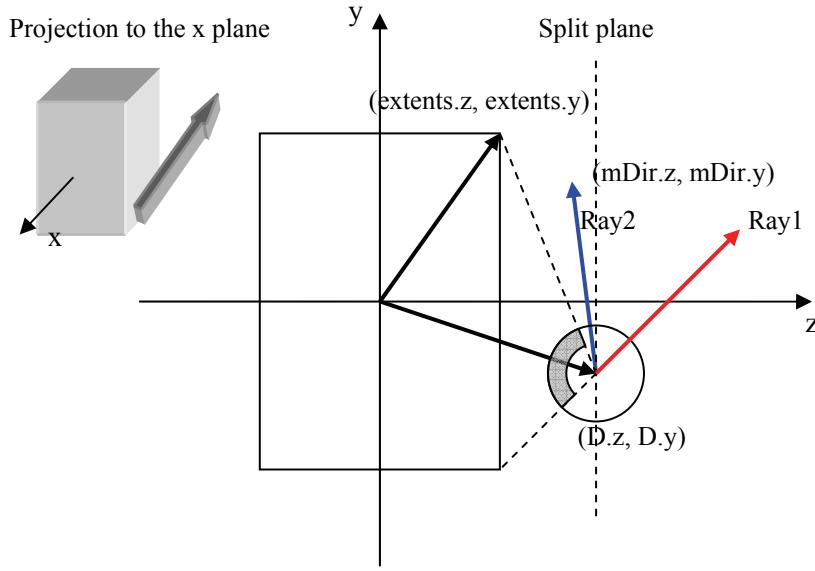


Figure 7.1 Ray/AABB Collision Testing: projection to x plane

Let us now look at the math for the above algorithm.

Pre-compute:

$$D = m\text{Orig} - \text{center}$$

$$|m\text{Dir}| = \text{abs}(m\text{Dir})$$

Test 1:

If $(|D_x| > \text{extent}_x) \wedge (D_x \times m\text{Dir}_x \geq 0)$ return false

If $(|D_y| > \text{extent}_y) \wedge (D_y \times m\text{Dir}_y \geq 0)$ return false

If $(|D_z| > \text{extent}_z) \wedge (D_z \times m\text{Dir}_z \geq 0)$ return false

Test 2:

If $|m\text{Dir}_y \times D_z - m\text{Dir}_z \times D_y| > (\text{extents}_y \times |m\text{Dir}|_z + \text{extents}_z \times |m\text{Dir}|_y)$ return false

If $|m\text{Dir}_z \times D_x - m\text{Dir}_x \times D_z| > (\text{extents}_x \times |m\text{Dir}|_z + \text{extents}_z \times |m\text{Dir}|_x)$ return false

If $|m\text{Dir}_x \times D_y - m\text{Dir}_y \times D_x| > (\text{extents}_x \times |m\text{Dir}|_y + \text{extents}_y \times |m\text{Dir}|_x)$ return false

Final step: return true

7.1.2.2 Ray OBB Collision

Ray AABB collision from the previous section can be used for oriented bounding boxes (OBB) with minor changes. We just transform both the ray and the OBB to a new coordinate system whose center is at the OBB's center and axis aligned with the OBB. This will turn OBB to AABB. Then we can use the same algorithm to test ray-box collision. The

transformation from OBB to AABB requires an additional computation to calculate the inverse of the rotation matrix of OBB and transform the 3D ray vector by it.

Assume ray (mOrig, mDir) and OBB (center, extents, mRot). mRot is the rotation matrix.

Pre-compute:

$\text{Inverse}(\text{mRot}) = \text{Inverse of rotational matrix } \text{mRot} (\text{mRot}^{-1})$

$\text{RayNew} = ((\text{mOrig}-\text{center}) \times \text{Inverse}(\text{mRot}), \text{mDir} \times \text{Inverse}(\text{mRot}))$

$\text{AABB} = ((0,0,0), \text{extents})$

Compute:

return using RayNew and AABB collision test.

We can optimize the above code by the fact that the inverse of a pure rotation matrix is the same as the transpose of the matrix. Hence we have

$$(\text{mRot}_{3\times 3}^{-1} = \text{mRot}_{3\times 3}^T) \wedge ((M_{3\times 3} \times V_{3\times 1})^T = V_{3\times 1}^T \times M_{3\times 3}^T) \Rightarrow V_{1\times 3} \times \text{mRot}_{3\times 3}^{-1} = (\text{mRot}_{3\times 3} \times V_{3\times 1})^T$$

Thus we can avoid calculating the inverse of the rotation matrix and modify the algorithm to the following.

Pre-compute:

$\text{RayNew} = (\text{mRot} \times (\text{mOrig}-\text{center}), \text{mRot} \times \text{mDir})$

$\text{AABB} = ((0,0,0), \text{extents})$

Compute:

return using RayNew and AABB collision test.

The optimization depends on a game engine's definition to the rotation matrix in OBB. If it is a 3×3 pure rotation matrix, then the modified version can be used. If it is not a pure rotation matrix, then we have to use the original one or something in the middle. The rotation matrix of OBB used in ParaEngine contains both a pure rotation matrix and a position translation. In other words, mRot is a 4×3 matrix.

7.1.2.3 Mouse Ray Picking

Mouse ray picking is to find the scene object that intersects with the user's mouse. In a 3D game, the screen position (x,y) of the mouse cursor is first converted to a 3D ray in world coordinate system using the current camera and projection settings; then this 3D ray will be used to pick the closest scene object that intersects with it.

The mathematics is fairly simple. Given a 2D position in screen coordinate system (x,y) with screen size (width, height) pixels, and a projection matrix (mProj) and camera view transform matrix (mView), we will compute the 3D ray with origin and direction (vOrg, vDir) as follows:

Compute the intersection point (vPtScr) between the ray and the view frustum's near plane in screen space as

$vPtScr = (2 \times x/\text{width}-1, -(2 \times y/\text{height}-1), Z_{\text{near}})$, where Z_{near} is the Z-value of the near view-plane.

Transform $vPtScr$ back to the camera view coordinate system using the inverse of $mProj$.

$vPtView = vPtScr \times mProj^{-1}$, where the projection matrix $mProj$ has the following format:

$$mProj = \begin{matrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & Z_{\text{far}}/(Z_{\text{far}} - Z_{\text{near}}) & 1 \\ 0 & 0 & -Z_{\text{near}} \times Z_{\text{far}}/(Z_{\text{far}} - Z_{\text{near}}) & 0 \end{matrix}$$

We can directly compute $vPtScr \times mProj^{-1}$ as

$$vPtView = (vPtScr_x / xScale \times Z_{\text{near}}, vPtScr_y / yScale \times Z_{\text{near}}, Z_{\text{near}})$$

Or we can use $vPrView' = vPrView / Z_{\text{near}} = (vPtScr_x / xScale, vPtScr_y / yScale, 1)$ which is also a point on the ray.

Now that we have two points on the ray in the view coordinate system, which is the camera origin $(0,0,0)$ and $vPtView$. Please note that $vPtView$ is also the direction of the ray. Thus we can compute the origin and direction of the final ray in world coordinate system by applying the inverse of the camera view matrix to them. Thus we have

Pre-compute:

$$mView^{-1} = \text{inverse of } mView$$

Compute:

$$vOrg = (0,0,0) \times mView^{-1} = (mView^{-1}.41, mView^{-1}.42, mView^{-1}.43)$$

The following code shows how a mouse position in 2D screen coordinates can be converted to a 3D ray in the world coordinate system. Because it is fairly simple, we will show directly by code.

```
// nWidth and nHeight is the screen size.

void CAutoCamera::GetMouseRay(D3DXVECTOR3& vPickRayOrig, D3DXVECTOR3& vPickRayDir,
POINT ptCursor, UINT nWidth, UINT nHeight)
{
    D3DXMATRIX* pMatProj = GetProjMatrix();

    // Compute the vector of the pick ray in screen space
    D3DXVECTOR3 v;
    v.x = (((2.0f * ptCursor.x) / nWidth) - 1) / pMatProj->_11;
    v.y = -((2.0f * ptCursor.y) / nHeight) - 1) / pMatProj->_22;
    v.z = 1.0f;

    // Get the inverse of the composite view and world matrix
    D3DXMATRIX* pMatView = GetViewMatrix();
    D3DXMATRIX m;
    m = (*pMatView);
    D3DXMatrixInverse( &m, NULL, &m );
}
```

```

// Transform the screen space pick ray into 3D space
vPickRayDir.x = v.x*m._11 + v.y*m._21 + v.z*m._31;
vPickRayDir.y = v.x*m._12 + v.y*m._22 + v.z*m._32;
vPickRayDir.z = v.x*m._13 + v.y*m._23 + v.z*m._33;
vPickRayOrig.x = m._41;
vPickRayOrig.y = m._42;
vPickRayOrig.z = m._43;
D3DXVec3Normalize(&vPickRayDir,&vPickRayDir);
}

```

7.2 Architecture and Code

Recall that the common base object for scene objects contains a virtual function which returns the OBB of its bounding volume. The scene manager can use it to quickly get the scene object that intersects with a ray. And we can start ray query with a certain distance. Objects whose distance to ray origin is larger than it will be neglected. And we provide a user defined object filter object that could reject objects according to some criteria, such as type and name.

7.2.1 Ray Picking in the Scene

The following shows the interface for object level ray picking in ParaEngine

```

/** it stands for an object intersecting with a ray. it is used for ray picking. */
struct PickedObject:public AttachedSceneObject
{
    /// approximated distance
    float m_fRayDist;
    /// the smallest value of the object's bounding box's extents
    float m_fMinObjExtent;
public:
    PickedObject(float fRayDist, float fMinObjExtent, CBaseObject* pObj)
        :m_fRayDist(fRayDist), m_fMinObjExtent(fMinObjExtent), AttachedSceneObject(pObj){}
    PickedObject():m_fRayDist(0), m_fMinObjExtent(0), AttachedSceneObject(NULL){}
};

bool PickingFilterMesh(CBaseObject* obj){
    ObjectType t = obj->GetMyType();
    return ( t!= CBaseObject::BipedObject);
}

bool PickingFilterBiped(CBaseObject* obj){
    ObjectType t = obj->GetMyType();
    return ( t== CBaseObject::BipedObject);
}

bool PickingFilterNotPlayer(CBaseObject* obj){
    return ( CGlobals::GetScene()->GetCurrentPlayer() != obj);
}

/**
 * Pick object using view clipping object.
 * pick the smallest intersected object which is un-occluded by any objects. Object A is considered
 * occluded by object B only if
 * (1) both A and B intersect with the hit ray.
 * (2) both A and B do not intersect with each other.
 * (3) B is in front of A, with regard to the ray origin.
 *
 * this function will ray-pick any loaded scene object(biped & mesh, but excluding the terrain) using their
 * oriented bounding box. A filter function may be provided to further filter selected object. this function will
 * transform all objects to render coordinate system. This will remove some floating point inaccuracy near

```

the camera position. Hence this function is suitable for testing object near the camera eye position. This function does not rely on the physics engine to perform ray-picking.

- * @see Pick().
- * @param ray: the ray in world coordinate system
- * @params pPickedObject: [out] the scene object that collide with the mouse ray. This may be NULL, if no object is found.
- * @params fMaxDistance: the longest distance from the ray origin to check for collision. If the value is 0 or negative, the camera view culling radius is used as the fMaxDistance.
- * @param pFnctFilter: a callback function to further filter selected object. if it is NULL, any scene object could be selected.
- * @return :true if an object is picked.
- */

```
bool CSceneObject::PickObject(const CShapeRay& ray, AttachedSceneObject* pTouchedObject, float fMaxDistance, OBJECT_FILTER_CALLBACK pFnctFilter)
```

The following code shows an implementation of picking object by traversing the scene graph using the bounding volume of the segmented ray.

```
bool CSceneObject::PickObject(const CShapeRay& ray, AttachedSceneObject* pTouchedObject, float fMaxDistance, OBJECT_FILTER_CALLBACK pFnctFilter)
{
    if(fMaxDistance<=0)
        fMaxDistance = m_sceneState.fViewCullingRadius;
    PickedObject lastObj;

    CRayCollider rayCollider;
    rayCollider.SetMaxDist(fMaxDistance);
    D3DXVECTOR3 vRenderOrigin=GetRenderOrigin();
    // the ray in the view space, shifted to the render origin.
    CShapeRay ray_view(ray.mOrig-vRenderOrigin, ray.mDir);

    // defining the view frustum based on the bounding sphere of the ray
    float fViewRadius = fMaxDistance/2;
    D3DXVECTOR3 vViewCenter = ray.mOrig + ray.mDir*fViewRadius;

    While (for each objects in the view frustum){
    ...// code omitted
        CBaseObject* pObj = queueNodes.front();
        CBaseObject* pViewClippingObject = pObj->GetViewClippingObject();

        queueNodes.pop();
        ObjectType oType = pObj->GetMyType();

        D3DXVECTOR3 vObjCenter;
        pViewClippingObject->GetObjectCenter(&vObjCenter);
        vObjCenter -= vRenderOrigin;

        // float fR = pViewClippingObject->GetBoundSphere();
        // rough testing using bounding sphere
        if(pViewClippingObject->TestCollisionSphere(&(vViewCenter), fViewRadius,1) ){
            // further testing using bounding box
            CShapeOBB obb;
            pViewClippingObject->GetOBB(&obb);
            obb.mCenter-=vRenderOrigin;
            float fDist;
            if(rayCollider.Intersect(ray_view, obb, &fDist) && fDist<=fMaxDistance
            /* filter objects*/&& (pFnctFilter==NULL || pFnctFilter(pObj)) ){
                // add to collision list and sort by distance to ray origin
            ...// code omitted
        }
    }
}
```

```

    }
}

if(pTouchedObject!=0 ){
    if(lastObj.IsValid()){
        pTouchedObject->m_pObj = lastObj.m_pObj;
        pTouchedObject->m_pTerrain = lastObj.m_pTerrain;
    }
    else
        pTouchedObject->m_pObj = NULL;
}
return false;
}

```

7.2.2 Collision Detection Math Code

The code presented here is based on an open source physics engine called ODE. It provides data structures for ray, AABB, OBB, etc, as well as collider classes for performing collisions between one shape and all other shapes. The reason that a collider class is used is that it may pre-calculate data that is shared among multiple collision queries. For example, if one uses the same ray to test collisions with 1000 objects, it does make sense to reuse the same collider which will save some CPU cycles.

```

class CRayCollider : public Collider{
    bool Collide(const CShapeRay& world_ray, const CShapeAABB& world_AABB, const
    bool Collide(const CShapeRay& world_ray, const CShapeOBB& world_oob);
    bool Intersect(const CShapeRay& world_ray, const CShapeAABB& world_AABB, float * pDist, const
D3DXMATRIX* world=NULL);
    bool Intersect(const CShapeRay& world_ray, const CShapeOBB& world_oob, float * pDist);
    void SetMaxDist(float max_dist);

    bool RayAABBOverlap(const D3DXVECTOR3& center, const D3DXVECTOR3& extents);

    bool SegmentAABBOverlap(const D3DXVECTOR3& center, const D3DXVECTOR3& extents);
    bool RayTriOverlap(const D3DXVECTOR3& vert0, const D3DXVECTOR3& vert1, const
D3DXVECTOR3& vert2);

    bool InitQuery(const CShapeRay& world_ray, const D3DXMATRIX* world=NULL);
protected:
    // Ray in local space
    D3DXVECTOR3 mOrigin; //!< Ray origin
    D3DXVECTOR3 mDir; //!< Ray direction (normalized)
    D3DXVECTOR3 mFDir; //!< fabsf(mDir)
    D3DXVECTOR3 mData, mData2;
    ...// code omitted
};

bool CRayCollider::InitQuery(const CShapeRay& world_ray, const D3DXMATRIX* world)
{
    // Reset stats & contact status
    Collider::InitQuery();

    // Compute ray in local space
    // The (Origin/Dir) form is needed for the ray-triangle test anyway (even for segment tests)
    if(world){
        Matrix3x3 InvWorld(*world);
        mDir = InvWorld * world_ray.mDir;

        D3DXMATRIX World;

```

```

CMath::InvertPRMatrix(World, *world);
D3DXVec3TransformCoord(&mOrigin, &world_ray.mOrig, &World);
}else{
    mDir = world_ray.mDir;
    mOrigin = world_ray.mOrig;
}

// Precompute data (moved after temporal coherence since only needed for ray-AABB)
if(IR(mMaxDist)!=IEEE_MAX_FLOAT){
    // For Segment-AABB overlap
    mData = 0.5f * mDir * mMaxDist;
    mData2 = mOrigin + mData;

    // Precompute mFDir;
    mFDir.x = fabsf(mData.x);
    mFDir.y = fabsf(mData.y);
    mFDir.z = fabsf(mData.z);
}else{
    // For Ray-AABB overlap

    // Precompute mFDir;
    mFDir.x = fabsf(mDir.x);
    mFDir.y = fabsf(mDir.y);
    mFDir.z = fabsf(mDir.z);
}
return false;
}

bool CRayCollider::SegmentAABBOverlap(const D3DXVECTOR3& center, const D3DXVECTOR3& extents){
float Dx = mData2.x - center.x; if(fabsf(Dx) > extents.x + mFDir.x) return false;
float Dy = mData2.y - center.y; if(fabsf(Dy) > extents.y + mFDir.y) return false;
float Dz = mData2.z - center.z; if(fabsf(Dz) > extents.z + mFDir.z) return false;
float f;
f = mData.y * Dz - mData.z * Dy; if(fabsf(f) > extents.y*mFDir.z + extents.z*mFDir.y) return false;
f = mData.z * Dx - mData.x * Dz; if(fabsf(f) > extents.x*mFDir.z + extents.z*mFDir.x) return false;
f = mData.x * Dy - mData.y * Dx; if(fabsf(f) > extents.x*mFDir.y + extents.y*mFDir.x) return false;
return true;
}

bool CRayCollider::RayAABBOverlap(const D3DXVECTOR3& center, const D3DXVECTOR3& extents){
float Dx = mOrigin.x - center.x; if(GREATER(Dx, extents.x) && Dx*mDir.x>=0.0f) return false;
float Dy = mOrigin.y - center.y; if(GREATER(Dy, extents.y) && Dy*mDir.y>=0.0f) return false;
float Dz = mOrigin.z - center.z; if(GREATER(Dz, extents.z) && Dz*mDir.z>=0.0f) return false;
float f;
f = mDir.y * Dz - mDir.z * Dy; if(fabsf(f) > extents.y*mFDir.z + extents.z*mFDir.y) return false;
f = mDir.z * Dx - mDir.x * Dz; if(fabsf(f) > extents.x*mFDir.z + extents.z*mFDir.x) return false;
f = mDir.x * Dy - mDir.y * Dx; if(fabsf(f) > extents.x*mFDir.y + extents.y*mFDir.x) return false;
return true;
}

bool CRayCollider::Collide(const CShapeRay& world_ray, const CShapeAABB& world_AABB, const
D3DXMATRIX* world){
    // Init collision query
    // Basically this is only called to initialize precomputed data
    if(InitQuery(world_ray, world)) return true;

    D3DXVECTOR3 vCenter, vExtents;
}

```

```

world_AABB.GetCenter(vCenter);
world_AABB.GetExtents(vExtents);

// Perform stabbing query
if(IR(mMaxDist)!=IEEE_MAX_FLOAT){
    return SegmentAABBOverlap(vCenter, vExtents);
}else{
    return RayAABBOverlap(vCenter, vExtents);
}

return true;
}

bool CRayCollider::Collide(const CShapeRay& world_ray, const CShapeOBB& world_oob){
    CShapeRay ray(world_ray);
    ray.mOrig -= world_oob.GetCenter();
    return Collide(ray, CShapeAABB(D3DXVECTOR3(0,0,0), world_oob.GetExtents()),
        &world_oob.GetRot());
}

```

7.3 Summary and Outlook

In the old days, ray picking is used widely in game engines to query physical information about the game world. In recent years, picking functions have been leveraged in specialized game physics modules. However, object level picking is still an indispensable part in modern computer game engine. The picking function discussed in this chapter is mainly used for object level mouse ray picking in ParaEngine.

Ray picking in triangle level may be implemented by different scene object independently. For example, if a ray hits an object, one can further calls the object's picking function to obtain sub-level collision information in the triangle level. BSP used to be a great data structure for performing triangle level ray picking on large level geometry. However, it is not efficient for collision detection of a great number of smaller game objects of other shapes such as sphere and box. Recent physics engine usually uses octree and/or quad tree combined with AABB to provide collision detection of virtually any shape at both triangle level and object level.

Caching is also a technique which could accelerate picking and other collision detection queries when they are issued in large qualities and that the collision spots are not far from one another. In simpler words, if in former queries, two objects are in contact with each other, they tend to continue to be so in subsequent queries; hence we can accelerate collision detection by keeping this information from query to query. Cached information should be invalidated and updated when object moves in the scene. One can find more information about picking and game physics on the web using the key words “game physics”.

Chapter 8 Simulating the Game World

There are always moving objects in a game world. They are not all movies, in which all movements are pre-made by artists; they are not all controlled by human users or intelligent computer agents. Instead, a game world usually has its own physical laws which automatically produces or constrains the motions of game objects in it. Due to the limited power of a personal computer, not every object can be simulated accurately as in our real world. However, the trend of next generation game engine is on the integration of more advanced game world simulation technologies and artificial intelligence.

One of the main tasks of simulating the game world is dealing with physics in it. We call it game physics or physics engine in the game industry. The most basic task of game physics is to ensure that two solid objects do not run into each other, as well as simulating forces on objects, such as gravity and thrust.

The focus of the chapter is on *simulation*, which involves the use of rigid body dynamics simulation, motion blending of key framed animations, as well as other methods to simulate the motions of objects in a game world. We will also examine the related modules in ParaEngine.

8.1 Foundation

This section will cover several physics simulation methods. They are rigid body dynamics, sensor based physics, motion blending, and autonomous character animation. They are usually used together in a computer game engine.

8.1.1 Rigid Body Dynamics

In recent years, game physics have come to the middleware arena. They are called physics engine, which provides APIs to simulate the movements of 3D geometry under effects of gravity, friction, collision with other objects, and even soft body motions such as cloth. But the selling point of most physics simulation middleware is on the efficient and robust implementation of rigid body dynamics. We do not intend to cover the math here. Interested reader can refer to SIGGRAPH '97 course notes on physically based modeling: principles and practice⁸. Understanding the math of rigid body dynamics is one thing, implementing them efficiently is another thing. Our suggestion is that always consider using a middleware first.

8.1.1.1 Integrating a Physics Engine

Two critical problems in integrating physics engine into a game engine are simulation time management and collision response. Simulation time is the current time used in the physics engine. Each frame, we advance simulation time in one or several steps until it reaches the current rendering frame time. Choosing when in the game loop to advance simulation and by how much can greatly affect rendering parallelism.

⁸ SIGGRAPH '97 course notes: <http://www.cs.cmu.edu/~baraff/sigcourse/>

In game development, we usually have many hand-animated (key framed) game characters, complicated machines and some moving platforms. These objects do not obey the laws of physics. They are non physical objects, but they should appear to be physical. So the problem is how physical and non-physical object should react to each other. For example, we may consider key framed motion to be nonnegotiable. A key framed sliding wall can push a character, but a character cannot push a key framed wall. Key framed objects participate only partially in the simulation; they are not moved by gravity, and other objects hitting them do not impart forces. They are moved only by key framed animation data. For this reason, the physics engine usually provides a callback function mechanism for key framed objects to update their physical properties at each simulation step. And game developers need to manually tell the engine how the objects should respond.

8.1.1.2 Novodex Physics Engine

There are several commercial and open source middleware physics engine today. Among the most popular are havoc, novodex, and ODE. Novodex is even forward looking to support hardware accelerated physics calculation. It has a public license for developers to try for free. We will demonstrate the use of Novodex engine to perform geometry level physics in ParaEngine.

A full integration guide of the Physics engine can be found here⁹. All physics SDKs are built upon a few fundamental objects: the physics SDK itself, a physics environment or scene, parameters for the scene, rigid bodies, materials that define the surface properties of the rigid bodies, collision shapes that the rigid bodies are built from, triangle meshes that some collision shapes are comprised of, constraints that connect rigid bodies at a point, and springs that connect rigid bodies via forces over a distance. The final component of the SDK is a set callback objects that developers can use to receive collision events, such as contact events between designated contact pairs, joint breakage, and the results of ray cast queries in previous frames, etc.

We call all physical objects in Novodex “actors” and use the words “actor” and “object” interchangeably. These actors can be crates, doors, trees, rocks, any physical object that resides in the game world. There are two types of actors: “dynamic” and “static”.

“Dynamic” actors are part of the physical scene and will move and collide with the world and other objects realistically. In other words, they are moved by force. A dynamic actor can be turned into a kinematic actor, if we want total control over them. A kinematic actor does not react to the physical world; yet the physical world reacts to kinematic actors. For example, they can be moved to wherever we want them, pushing all dynamic objects aside along the way, as if they had infinite mass. They are good for moving objects that are effectively immune to the physical scene, like Main RPG character, heavy moving platforms or large moving blast doors and gates, etc. Kinematic objects can be turned into dynamic objects and vice-versa.

“Static” actors are stationary and never move in the scene. Static objects cannot be made dynamic or kinematic once they are created.

⁹ NovodeX AG, NovodeX SDK Integration Guide, www.novodex.com, 2005.

To summarize, the physical world is built with dynamic and static actors. Two kinds of dynamic actors are interchangeable; they are dynamic actors and kinematic actors. Kinematic actor can be thought of as a mobile static object, as it will push other dynamic objects aside. In the following text, we will use static actor, dynamic actor and kinematic actor to distinguish these three types of actors which comprise the physical world.

Kinematic Character Collision Detection and Response

The kinematic motion of character is calculated after the dynamic simulation of the physical environment is finished, which in turn will affect the dynamic objects in the next time step. We will first look at the simulation pipeline in ParaEngine.

The Environment Simulation Routine is activated several times a second (may be 15 or 30 FPS). The physics engine (in this case, it is the Novodex engine) usually has a higher frame rate (usually 50 or 60 FPS) for precise collision detection; hence in each environment simulation step, the Novodex engine may advance several sub steps until it catches up with the simulation time, which in turn may need to catch up the rendering time. During each sub steps, the physics engine may report user contacts through callbacks (since, Novodex is multi-threaded; these callbacks are in different threads than the game loop thread). The game engine should remember these contacts in the callbacks; so that it can see them and respond to them when the game loop thread resumes control.

The following code gives an overview of what should be done in the game loop thread.

```
Environment simulation (SIM_TIMER) {  
    Fetch last simulation result of dynamic objects  
    Validate simulation result and update scene object parameters, accordingly.  
    Calculate kinematic object motions and update simulation data for the next time step.  
    Run AI module (SIM_TIMER) {  
        Run scripting system (SIM_TIMER):  
            Networking is handled transparently through the scripting system.  
    }  
    Start simulating for the next time step (this may run in a separate thread than the game loop).  
}
```

The motion of kinematic objects are usually calculated from sensor results, such as casting rays in front of the character and returning contact points with the physics environment. One advantage of using sensors is that one can detect collision before it happens. The shape of the sensors used in collision detection can be point, line or even the bounding shape of the character itself.

Terrain learning is required for collision prevention. However, this task is less realistic because it relies on trial and error rather than prediction, and that a character will only detect obstacles when it is too late (i.e. already in contact).

8.1.2 Sensor Ray based Physics

Physics in a game can be completed based on sensor rays or through other physics simulation techniques, such as the rigid body dynamics. This section will focus on the first one. In **Chapter 7**, we have shown that ray picking is a common feature supported by different kinds of scene objects, such as bounding volumes, triangle mesh, terrain and ocean. A physics simulation engine which is entirely based on ray casting can be devised. In case that both the environment and the object being simulated is extremely complex, ray picking is the only choice left. In role playing games and first person shooter games, sensor ray based physics are still popular for the main game character. We can also use sensor rays for handling other kinematics character collision detections and responses.

The idea of sensor ray based physics is similar to using our eye to perceive the environment. However, the process is reversed. Instead of letting the light rays coming into our eyes, we actively cast sensor rays to the environment and deduct the environment from information returned from sensor rays. Obviously, the more rays (higher resolution) cast, the more accurate environment information a program can deduct. The disadvantage of ray based physics is that no matter how many rays we cast, we can only deduct and never be sure about the space between two adjacent rays. The advantage is that it is a common way to perceive the environment. One can use the same algorithm for environment of any shape and of any complexity.

In ParaEngine, character objects cast multiple sensor rays to get the knowledge about physical environment in its surroundings and animate accordingly. We will use it as an example of showing how sensor rays can be used in game physics. In our implementation, the character is modeled as a cylinder will move in a 3D scene with meshes, terrain and ocean. It will walk on land, swim in water, fall in air, climb on slopes, slide if there is a wall, and block if there is a corner, etc. All this can be done almost faultlessly and smoothly using only ray based physics.

Character's Physical Definition

- dTimeDelta: the time delta to be simulated. Character's position and speed will be updated according to this time step. Such as 1/30 second.
- vPos: 3D position of the object in world coordinate system. Y is for height (up) axis.
- vPosTarget: the destination position that the character is instructed to move. The Y component of vPosTarget is ignored, assuming the character should always be on ground.
- Speed: the magnitude of the character's current speed vector after projection to the X,Z planes.
- vFacing: the direction of character's current speed. It does not count for the vertical speed.
- SpeedVertical: the magnitude of the character's current vertical speed (i.e. speed in Y plane).
- Radius: the character is modeled as a cylinder. This is the radius of this cylinder. We also call it physical radius of the character. Such as 0.5 meters
- Height: height of the character. Also the cylinder's height. We also call it physical height of the character. Such as 1.8 meters.

Constants

- PENETRATION_DISTANCE: the maximum penetration distance allowed. In each simulation step, if the biped moves longer than this step, it is further divided into several evenly spaced steps. The step length is determined by the biped's speed multiplied by the time delta. Default value 0.1 meter.
- MAX_STEPS_PER_FRAME: ignore PENETRATION_DISTANCE, if it takes longer than MAX_STEPS_PER_FRAME to complete in a simulation frame. This is likely to happen when character have very fast speed. This will ensure that the physics simulation will always complete within a certain sub steps. However, it may produce inaccurate physics.
- BIPED_SENSOR_RAY_NUM: The number of rays in a sensor group. The higher this number, the more accurate the collision detection will be. The value must be 2n+1, such as 1,3,5,7,9,etc. The direction of the i^{th} ray is given by the following equation:

$$i^{\text{th}} \text{ ray direction} = \text{Object facing} + (\text{PI} / (\text{rayNum}+1)) * (i - ((\text{rayNum}-1) / 2))$$

Normally 3 rays will be both accurate and fast enough for most situations of RPG characters. See Figure 8.1.

- FALLDOWN_SPEED: the vertical speed at which the biped will fall down from high ground. Default value is 4.0.
- CLIMBUP_SPEED: the vertical speed at which the biped will jump up on a shallow stair. Default value is 2.0.
- REBOUND_SPEED: the speed at which the biped will rebound from a wall, assuming that the mass of characters are the same. This is proportional to rebound impulse. Default value is 0.3.
- WALL_REBOUNCE_DISTANCE: the biped will rebound from a wall, if its body penetrates into the wall for the specified distance. Default value is 0.4.
- SENSOR_HEIGHT_RATIO: the ratio between the sensor ray height and the biped's physical height. Default value is 0.5.
- GRAVITY_CONSTANT: gravity. Value is 9.81.

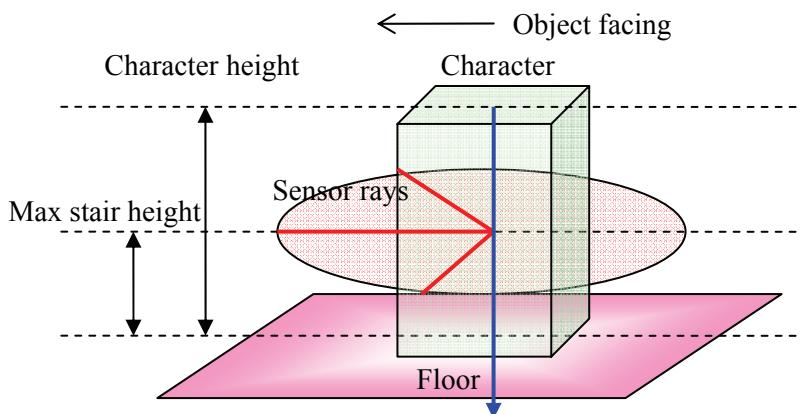


Figure 8.1 Multiple ray-casting collision detection

The origin of all sensor rays is the character's global position plus a max stair height, which is usually some factor of the character's height. A character will automatically climb up stairs lower than this height.

Collision Response and Wall Sliding

We will first show collision detection and wall sliding without worrying about the terrain and water, etc. In the multiple-ray-casting method, the character avoids obstacles using the sensor ray hit points for detecting areas of free space instead of solely relying on the normal of the obstacles. The obstacle normal returned by the ray sensor query is only used to give a tentative move direction. Then we shift the origin of the sensor ray to the new location and query again to obtain another hit point. From the two hit points together with the velocity vector of the character, we calculate another impact normal, which will be used to find an alternative heading. Using hit points of sensor rays to obtain impact normal can be an advantage, because using the obstacle normal to find free space is a rough estimate, and the likelihood of success diminishes as the complexity of the environment increases.

The algorithm is given below:

```
vMovePos = vPos  
vOrig0 = (vPos.x, vPos.y + Height * SENSOR_HEIGHT_RATIO, vPos.z)  
vDir0 = vFacing  
Range0 = Radius  
Cast a group (group0) of n (n=3) rays in front of the character, the middle ray in the group has  
origin (vOrig0), direction (vDir0), and sensor range(Range0)  
  
If several sensor rays hit some obstacles within the Radius of the object{  
    if the world impact normals are roughly the same{  
        /* the character is considered to be blocked by a single wall. In case of a single  
        blocking wall, we will try slide the character along the wall; */  
        vWallNorm = average of all impact normals returned by the ray query.  
        // compute a tentative sliding wall facing(not the final one) as below:  
        vTentativeFacing = vWallNorm × (vFacing × vWallNorm)  
  
        vOrig1 = vOrig0 + vTentativeFacing × PENETRATION_DISTANCE  
        vDir1 = the direction of previous ray whose hit point is closest to vOrig0  
        Range1 = Radius + Speed × dTimeDelta  
        Cast another ray1 (vOrig1, vDir1, Range1)  
  
        Bool bMoveAlongOldFacing=false  
        If (the ray hits anything){  
            // get the wall direction by two hit points of the sensor ray  
            vWallDir = ray1.hitpoint - group0.hitpoint  
            if(vWallDir is not (0,0,0)){  
                /*check if the sensor ray and the biped facing vector are on the same side of the  
                wall vector. If so, it means that the biped is currently walking into the wall,  
                otherwise it is leaving the wall. */  
                If( (vDir1 × vWallDir) • (vFacing × vWallDir) > 0 ){
```

```

        speedScale = vFacing • vWallDir
        vMovePos=vPos+vWallDir × (Speed × dTimeDelta × speedScale)
    }else{
        bMoveAlongOldFacing = true
    }

}else{
    bMoveAlongOldFacing = true
}

}else{
    bMoveAlongOldFacing = true
}
}

If(bMoveAlongOldFacing is true){
    /*we permit walking along the old facing vector. This is the case when the
character is walking away from the wall.*/
    vMovePos=vPos+vFacing × (Speed × dTimeDelta)
}

/** Finally, we will move the character slightly out of the wall, if it has run too deep in
to it.*/
if(vMovePos != vPos){
    vOrig2 = vOrig0+(vMovePos-vPos)
    vDir2 = vFacing
    Range2 = Radius
    Cast a group (group2) of n (n=3) rays using the new character position

    If(group2 hit anything){
        nRes=compare sensor ray group0 and group2 by their hitpoints to orig distances
        if(bMoveAlongOldFacing is false and nRes>0){
            // The character is cornered; restore it to its original position
            vMovePos = vPos
        }else{
            ReboundDistance=Radius-Groups2. MinDist-PENETRATION_DISTANCE;
            If(ReboundDistance>=0){
                // move character slightly out of wall
                vMovePos=vMovePos+Group2.vDir × (-ReboundDistance)
            }
        }
    }
}else{
    Character is considered to be at a corner. Stop the character.
}
}else{
    // if nothing is hit, simply move the character forward
    vMovePos=vPos+vFacing × (Speed × dTimeDelta)
}

vMovePos now contains the position of character after collision detection and wall sliding in
the x,z plane.

```

Figure 8.2 shows the above algorithm. The character's facing, wall obstacle and the origins of sensor ray group 0, 1, 2 are marked out. The character will finally slide along the wall to the bottom left.

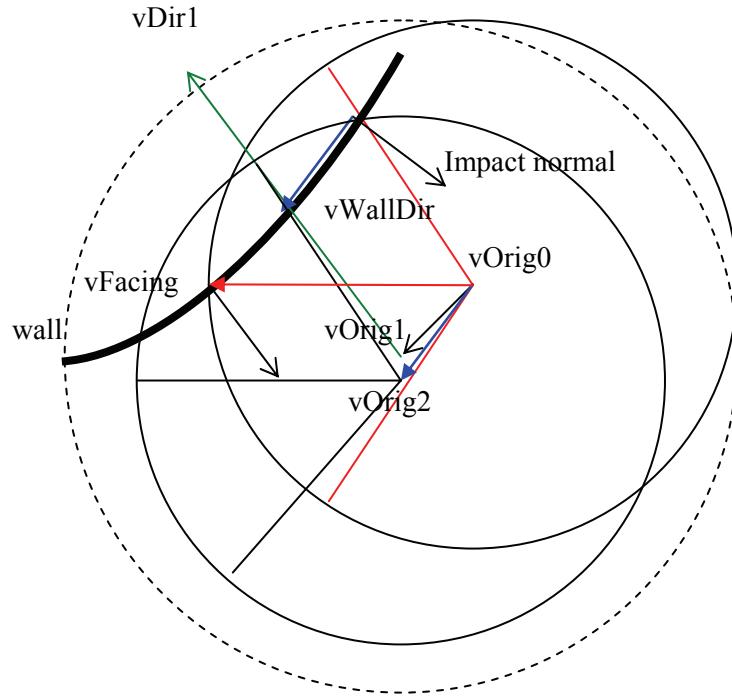


Figure 8.2 Ray casting based sliding wall

The Complete Code

The complete code can be found in the code section later in this chapter. It takes other factors such as terrain height, water level, etc in to account.

The Implementation Result

The game demo on our website uses the algorithm described in this section. Figure 8.3 shows characters sliding along small cylinder physics object, climbing up stairs, sliding along wall meshes, turning around sharp wall edges, etc. The algorithm will generate accurate and smooth motion even with concave, convex and sharp physical meshes.



Figure 8.3 Collision detection and response results

8.1.3 Motion Blending

Realistic character simulation is one of the hardest simulations in game engine; because a character usually has dozens of joints which move together in complex patterns and with complex constraints. So character simulation in current game engine implementation is usually achieved by motion blending premade animations according to physical events and user/script commands. Premade animations are traditional key frame based animations which are made by artists, frame by frame and/or using motion capture devices. For example, a character has key framed animations for walking, running, jumping, falling and attacking. A physical event generated by the physics engine, such as a sudden change on the terrain, might change the character state from walking to falling. The character will blend walking animation and falling animation with a blending factor, so that the motion of character is smoothly transformed from walking to falling.

Since a complete introduction to motion blending requires knowledge about character's skeleton system, we will cover it in the character animation **Chapter 12**. When motion blending is used in the simulation system, the character is modeled using a simple shape, such as cylinder, sphere or capsule, as well as a group of state parameters. The simulation module

can access the state parameters associated with a character object during simulation. For example some state parameters may indicate that the character is in air, on land or under water, etc. In the rendering pipeline, we will synthesize the final animation of a character from a pool of key framed animations and its associated state. This synthesis of motion is called motion blending.

State of the Art

As computer becomes faster, the simulation module usually replaces some animation parts which used to be made with key framed animation. For example, cloth, wave, particle simulation are moving to the physically based simulation module.

The animations generated by rigid body dynamics are more diverse and realistic than key framed or motion blended animation. However, at the moment, they are only realistic under limited conditions. For instances, rigid body dynamics are used to simulate character motions when they are considered dead in the game, such as person or animals shot dead or their corpses being dragged on the floor.

8.1.4 Autonomous Animation

The ultimate goal of character motion simulation is autonomous animation, which will learn by observation the patterns of character motions under different environment and having them automatically generated at playing time.

When animating a character, there are three kinds of animations which are usually dealt with separately in a motion synthesis system: (1) local animation, which deals with the motion of its major skeleton (including its global speed), (2) global animation, which deals with the position and orientation of the character in the scene, (3) add-on animation, which includes facial animation and physically simulated animation of the hair, cloth, smoke, etc. This chapter mainly talks about the local animation. Local animation is usually affected by the states of the character (such as a goal in its mind or a user command) and its perceptible vicinity (such as terrain and sounds).

The motion of a specified human character can be formulated by a set of independent functions of time, i.e. $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_0, +\infty]\}$. These functions or variables typically control over 15 movable body parts arranged hierarchically, which together form a parameter space of possible *configurations* or poses. For a typical animated human character, the dimension of the configuration is round 50, excluding degrees of freedom in the face and fingers. Given one such configuration at a specified time and the static attributes of the human figure, it is possible to render it at real-time with the support of current graphic hardware. Hence, the problem of human animation is reduced to, given $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_0, T_c]\}$ and the environment W (or workspace in robotics), computing $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_c, T_c + \Delta T]\}$ which should map to the desired and realistic human animation.

The first option to motion generation is to simply play back previously stored motion clips (or short sequences of $\{f_n(t)\}$). The clips may be key-framed or motion captured, which are used later to animate a character. Real-time animation is constructed by blending the end of one motion clip to the start of the next one. To add flexibility, joint trajectories are

interpolated or extrapolated in the time domain. In practice, however, they are applied to limited situations involving minor changes to the original clips. Significant changes typically lead to unrealistic or invalid motions. Alternatively, flexibility can be gained by adopting kinematic models that use exact, analytic equations to quickly generate motions in a parameterized fashion. Forward and inverse kinematic models have been designed for synthesizing walking motions for human figures¹⁰. There are also several sophisticated combined methods¹¹ to animate human figures, which generate natural and wise motions (also motion path planning) in a complex environment. Motions generated from these methods exhibit varying degrees of realism and flexibility.

A different motion generation framework aims at synthesizing real-time humanoid animation by integrating the variables of the environment into the controllers of the human body and using a learning and simulation algorithm to calculate and memorize its motion. Both the environment and the human body can be partially or fully controlled by an external user; whereas the motion for the uncontrolled portion will be generated from an internal algorithm. To produce realistic animation, the environment and the body movements are first fully controlled until the animation system has discovered the patterns for the various combinations of the different parts of the body and the environment variables; then only the environment and selected parts of the human body are controlled, the system will generate the motion for the rest. The advantages of the framework are (1) the motion is fairly realistic since it is based on examples. (2) different parts of the human body may act less dependently; e.g. the top of the body might react to other environmental changes other than synchronizing with the bottom of the body. Please refer to our paper¹² for more information.

8.2 Architecture and Code

The architecture of the simulation module involves the management of both static and dynamic objects.

(1) Static physics objects management: There are usually tens of thousands of physical mesh objects in a scene. It is not possible to simulate them concurrently. Fortunately, most physical mesh objects are static, so we can choose to simulate only objects in the vicinity of mobile physics objects. And a garbage collector routine will remove static physics objects which are far away from its nearest mobile object. Usually a reference counter will suffice to do the task. Another way to remove a bunch of physics objects from the simulator is through the managed loader class supported by the scene manager. Objects in a managed loader can be loaded on demand, yet unload as a single entity.

(2) Mobile object management: global characters are the most common mobile objects. The game engine must be able to simulate and render them very fast. If there are too many global objects (thousands), special data structures need to be used to store them according to their dynamic spatiality.

¹⁰ R. Boulle, D. Thalmann, and N. Magnenat-Thalmann. A global human walking model with real time kinematic personification. *The Visual Computer*, 6(6), December 1990.

¹¹ Kuffner, J. J., Autonomous Agents for Real-Time Animation. Ph.D. thesis, Stanford University, 1999.

¹² Xizhi Li, Synthesizing Real-time Human Animation by Learning and Simulation. 2004.

The module for simulation in ParaEngine is called Environment Simulator. It uses a physics middleware (Novodex physics engine) for handling static mesh collision detection and rigid body dynamic simulations.

8.2.1 Integrating the Physics Engine

We use a thin wrapper called CPhysicsWorld for wrapping the physics middleware. A scene manager is always associated with an instance of this class. It holds data for static physics meshes and rigid body dynamic objects. As the game progress, objects may be moved in and out of the physics world. The scene manager contains all objects in the scene. The physics world only contains a subset of objects in the scene, which are currently being simulated. When an object is deleted from the scene manager, it is also deleted from the physics world.

```
class CPhysicsWorld{
public:
    /** the triangle mesh shape. since we need to create separate physics mesh with different scaling
    factors even for the same mesh model, we will need the mesh entity plus the scaling vector to fully
    specify the mesh object*/
    class TriangleMeshShape {
public:
    MeshEntity* m_pMeshEntity; /// the mesh entity
    D3DXVECTOR3 m_vScale;   /// the scaling factor

    NxTriangleMeshShapeDesc m_ShapeDesc;
    TriangleMeshShape (){
        m_pMeshEntity = NULL;
        m_vScale = D3DXVECTOR3(1.f,1.f,1.f);
    }
};

/// All shapes. There may be multiple object using the same shape
list<TriangleMeshShape*> m_listMeshShapes;
public:
    NxPhysicsSDK*    m_pPhysicsSDK;
    NxScene*         m_pScene;
    NxVec3          m_pDefaultGravity;

    /** get a pointer to physics scene object */
    NxScene* GetScene(){return m_pScene;}

    /** get a pointer to physics SDK object */
    NxPhysicsSDK* GetPhysicsSDK(){return m_pPhysicsSDK;}

    /** Init the physics scene */
    void InitNx();

    /** ReleaseNx() calls NxPhysicsSDK::releaseScene() which deletes all the objects in the scene and
    then deletes the scene itself. It then calls NxPhysicsSDK::release() which shuts down the SDK.
    ReleaseNx() is also called after glutMainLoop() to shut down the SDK before exiting.*/
    void ReleaseNx();

    /** First call ReleaseNx(), then InitNx() */
    void ResetNx();

    /** this function will not return, until the rigid body physics had finished */
    void GetPhysicsResults();

    /** Start the physics for some time advances
}
```

```

    * @params dTime: advances in seconds */
void StartPhysics(double dTime);

<*/
* creat a static actor in the physical world from a mesh entity.
* @params pV: the world position of the actor
* @params fFacing: the facing around the y axis.
*/
NxActor* CreateStaticMesh(MeshEntity* ppMesh, const D3DXMATRIX& globalMat);

void ReleaseActor(NxActor* pActor);
...// omitted some functions
};

```

8.2.2 Environment Simulator

The main simulation module in ParaEngine is called environment simulator, which have been covered in **Chapter 1 and 2**. We will go for its details in this section. The entry function of the simulator is Animate(), which advances the simulation by a given time step. After each simulation, the environment simulator will generate intermediate data such as a list of active bipeds, each of which contains objects in its perceptive radius, etc.

```

Class CEnvironmentSim{
    /// a list of terrain that contains all active bipeds.
    list <ActiveTerrain*> m_listActiveTerrain;
    /// active bipeds may contain biped that is out of the view-culling radius.
    list <ActiveBiped*> m_listActiveBiped;
    /// very important biped list
    list <CBipedObject*> m_listVIPBipeds;
public:
    virtual void Animate( double dTimeDelta );
    void CleanupIntermediateData();
private:
    void Simulate(double dTimeDelta);
    void GenerateActiveBipedList();
    void GenerateStaticCP(ActiveBiped* pActiveBiped);
    void BipedSimulation();
    void PlayerSimulation(double dTimeDelta);

    /** when this function is called, it ensures that the physics object around an object is properly loaded.
    It increases the hit count of these physics objects by 1. The garbage collector in the physics world may
    use the hit count to move out unused static physics object from the physics scene (Novodex). This
    function might be called for the current player, each active mobile object in the scene and the camera
    eye position. */
    void CheckLoadPhysics(D3DXVECTOR3 vCenter, float fRadius);
...// omitted some functions and members
};

```

Active Biped is an important data structure that the environment simulator generates during each simulation step. An active biped structure contains reference to the character object, the terrain tile that contains this character, and a list of other characters in its perspective radius.

Recall that the scene manager maintains a global list of mobile characters. However, using this global list for collision detection and rendering is inefficient. Its computation complexity is $O(n^2)$, which tests one character against every other in the list. The environment simulator utilizes the quad-tree terrain tile available in the scene manager. It saves a temporary reference to each mobile character at the smallest quad-tree terrain tile that contains it. Recall that in **Chapter 4**, the terrain tile object has a member called `m_listBipedVisitors`, which is a

list of characters that are visiting that tile region. When character moves, the environment simulator will automatically update the list in affected terrain tiles. When rendering characters, only bipeds in the potentially visible terrain tile will be tested for drawing. And when a character tries to generate a list of visible characters in its vicinity, it will only search in the terrain tile that contains it as well as in its adjacent terrain tiles. In the worst case, there will be 9 terrain tiles that will need to be examined for generating visible character list for a given character.

```

//-----
// intermediate object definitions
//-----

struct ActiveTerrain{
    CTerrainTile* pTile;
    list <ActiveBiped*> listActiveBiped;
};

//-----
/// It holds any information that is perceived by a Active Biped object
/// including himself. This is one of the major product that environment
/// simulator produces for each living bipeds.
//-----

struct ActiveBiped{
    struct PerceivedBiped{
        /// the distance of the perceived biped to the host
        float fDistance;
        CBipedObject* pBiped;
        PerceivedBiped(float fDist, CBipedObject* pB){
            pBiped = pB;
            fDistance = fDist;
        };
        PerceivedBiped(){pBiped=NULL;};
    };
    /// the biped scene object.
    CBipedObject* pBiped;
    /// tile that tells most exactly where the object is in the scene
    /// the terrain in which the biped is in
    CTerrainTile* pTerrain;
    /// object list that collide with this biped. It can be solid object or other bipeds.
    list <CBaseObject*> listCollisionPairs;
    /// object list that this biped could see or perceive. this includes all object
    /// that is in the perceptive radius.
    list <PerceivedBiped*> listPerceptibleBipeds;
    ...// omitted some functions.
};

```

There is, however, a problem when the character is at the boundary of two or three adjacent terrain tiles. There are three solutions to it:

(1): increase the minimum size of terrain tiles that can store visiting biped lists and using the character's point position (not its bounding volume) to locate the terrain tile that contains it. Ignore the problem in other cases. For example, if the container terrain tile is 200*200 square meters, chances are small for characters walking on the boundary. And a special game title may even enforce that creatures can only move within a terrain tile but not crossing it.

(2): have duplicate references to the same character in terrain tiles that either contains or intersects with the character's perceptive volume. This method will work on all conditions, but is a little bit difficult to implement.

(3): use the character's point position (not its bounding volume) to locate the only terrain tile that contains it. During collision testing, the character should not only test against other characters in the container terrain tile, but also any adjacent tiles that intersects with its perceptible region (usually a sphere).

Another optimization we can apply is that we only need to update characters that are moving in the last simulation step. We can quickly detect if a character is moving or not by its speed magnitude. The following shows the character simulation pseudo code in the environment simulator.

Character simulation pseudo code

For more information about character simulation, please see the AI in Game World **Chapter 14**.

```
// Compute next scene for all game objects and generate the vicinity list.
CEnvironmentSim::Simulate(dTimeDelta){

    // Load physics around the current player and the camera position.
    // This is very game specific. Usually, it only ensures that physics object
    // around the current player and camera eye position is loaded.
    Call CheckLoadPhysics(player  position, player  radius*2)
    Call CheckLoadPhysics(camera  position, camera  near_plane*2)

    // Pass 1:Update Game Objects: building perceived object lists for each sentient object
    for each sentient game objects (sentientObj) in the scene{
        sentientObj.m_nSentientObjCount=0;
        sentientObj.m_PerceivedList.clear();

        update itself in the terrain tile according to its current position:

        for each valid terrain tiles(9 or more) in the neighbourhood of sentientObj{
            for each object (AnotherObj) in the terrain tile{
                if(AnotherObj is not a sentient object){
                    if sentientObj falls inside the sentient area of any other game object(AnotherObj){
                        wake up AnotherObj, add it to the back of the sentient object list.
                        AnotherObj.OnEnterSentientArea();
                    }
                }
                if AnotherObj falls inside the sentient area of sentientObj{
                    sentientObj.m_nSentientObjCount++;
                    if AnotherObj falls inside the perceptive area of sentientObj{
                        sentientObj.m_PerceivedList.push_back(AnotherObj.name());
                    }
                }
            }
        }
        if(sentientObj.m_nSentientObjCount==0){
            sentientObj.OnLeaveSentientArea();
            remove sentientObj from the sentient object list.
        }else{
            // call game AI now or in the next pass, we advise a second pass,
            // so it gives equal chance to each character
            // sentientObj.OnPerceived();
        }
    }

    // Pass 2 (It can be combined with Pass 1): call game AI of sentient objects
}
```

```

for each sentient game objects (pObj) in the scene{
    // generate way points
    pObj->PathFinding(dTimeDelta);
    // move the biped according to way point commands
    pObj->AnimateBiped(dTimeDelta);
    // apply AI controllers
    if(pObj->GetAIModule())
        pObj->GetAIModule()->FrameMove((float)dTimeDelta);
    if(!pObj->m_PerceivedList.empty())
        pObj->On_Perception();
    // call the frame move script if any.
    pObj->On_FrameMove();
}

// Animate all global particles
for each particle system (pObj) in the scene{
    if( !pObj->IsExploded() )
        pObj->Animate(dTimeDelta);
}
}

```

8.2.3 Sensor Ray Structure

It represents a group of rays sharing the same origin.

```

/** used for ray casting biped for low level navigation.
 * @see CBipedObject::MoveTowards()*/
struct SensorGroups{
    /// origin shared by all sensor rays in this group.
    D3DXVECTOR3 m_vOrig;
    struct SensorRay
    {
        /// direction of the ray in the x,z plane
        D3DXVECTOR3 vDir;
        /// the impact norm of the ray
        NxVec3 impactNorm;
        /// the hit point of the ray
        NxVec3 impactPoint;
        /// the distance from the impact point to origin.
        float fDist;
        /// whether the sensor has detected anything in its range
        bool blsHit;
    public:
        SensorRay():blsHit(false),fDist(0){};
    };
    /** n (n=BIPED_SENSOR_RAY_NUM=3) rays in front of the character, which covers a region of (n-
    2)/(n-1)*Pi radian.*/
    SensorRay m_sensors[BIPED_SENSOR_RAY_NUM];
    /// the average of all impact norms if available.
    D3DXVECTOR3 m_vAvgImpactNorm;
    /// range for all sensors.
    float m_fSensorRange;
    /// number of walls we hit.
    int m_nHitWallCount;
    /// the ray index whose impactPoint is closest to the origin. if negative, no ray has hit anything.
    int m_nHitRayIndex;

public:
    /** whether the sensor group has hit anything. */
    bool HasHitAnything(){ return m_nHitRayIndex>=0;}
}

```

```

/** whether the sensor group has hit anything. A negative value is returned if no hit ray.*/
int GetHitRayIndex(){ return m_nHitRayIndex;}
/** get the hit ray */
SensorRay& GetHitRaySensor(){return m_sensors[m_nHitRayIndex];}
/** get the number of wall hits. */
int GetHitWallCount(){ return m_nHitWallCount;}
/** get average impact norm. this is not normalized and may not be in the y=0 plane*/
D3DXVECTOR3 GetAvgImpactNorm(){ return m_vAvgImpactNorm;}

/** reset the sensor group to empty states. */
void Reset()
void ComputeSensorGroup(const D3DXVECTOR3& vOrig, const D3DXVECTOR3& vDir, float
fSensorRange, int nSensorRayCount = BIPED_SENSOR_RAY_NUM, float fAngleCoef = 0.16f);

bool CompareWith(const SensorGroups& otherGroup, float fAngleCoef = 0.16f);
...// omitted some functions.
};

```

8.2.4 Ray based Character Simulation

The following shows the actual implementation of the ray based character simulation discussed previously.

```

bool CBipedObject::MoveTowards( double dTimeDelta, const D3DXVECTOR3& vPosTarget, float
fStopDistance, bool * plsSlidingWall)
{
    static SensorGroups g_sensorGroups[3];
    float fMaxPenetration = (float)(m_fSpeed * dTimeDelta);
    if (fMaxPenetration>PENETRATION_DISTANCE){
        int nNumSteps= (int)ceil(fMaxPenetration/PENETRATION_DISTANCE);
        if(nNumSteps> MAX_STEPS_PER_FRAME)
            nNumSteps = MAX_STEPS_PER_FRAME;
        double dT = dTimeDelta/nNumSteps;
        bool bReachedDestination = false;
        for (int i=0;(i<nNumSteps) && (!bReachedDestination);i++)
        {
            bReachedDestination = MoveTowards(dT, vPosTarget, fStopDistance, plsSlidingWall);
        }
        return bReachedDestination;
    }
    float fGravity = GRAVITY_CONSTANT;
    {
        D3DXVECTOR3 vHeadPos = m_vPos;
        vHeadPos.y += GetPhysicsHeight();
        if(CGlobals::GetOceanManager()->IsPointUnderWater(&vHeadPos))
        {
            GetBipedStateManager()->AddAction(CBipedStateManager::S_IN_WATER);
            fGravity *= 0.2f;
            if(m_fSpeedVertical > 0)
#define UP_WATER_RESISTENCE_COEF 0.1f
#define DOWN_WATER_RESISTENCE_COEF 0.1f
                fGravity += m_fSpeedVertical*m_fSpeedVertical*UP_WATER_RESISTENCE_COEF;
            }
            else
            {
                fGravity -= m_fSpeedVertical*m_fSpeedVertical*DOWN_WATER_RESISTENCE_COEF;
            }
        }
    }
}

```

```

        GetBipedStateManager()->AddAction(CBipedStateManager::S_ON_FEET);
    }

    bool bReachPos = false;
    bool bSlidingWall = false; // whether the object is sliding along a wall.
    bool bUseGlobalTerrainNorm = false;
    bool bIgnoreTerrain = false;

    D3DXVECTOR3 vMovePos = m_vPos; // the position for the next frame without the height
    D3DXVECTOR3 vSub;
    float fDist;

    // get distance from target
    D3DXVec3Subtract( & vSub, & m_vPos, & vPosTarget );
    fDist = D3DXVec2LengthSq( & D3DXVECTOR2(vSub.x, vSub.z) );

    // check if we have already reached the position
    if(fStopDistance == 0)
    {
        if(fMaxPenetration * fMaxPenetration >= fDist)
        {
            // we're within reach
            bReachPos = true;
            // set the exact point
            vMovePos = vPosTarget;
        }
    }
    else if( fStopDistance > fDist)
    {
        /// if we're within reach, we will stop without reaching the exact target point
        bReachPos = true;
    }

    // get the biped's facing vector in y=0 plane.
    D3DXVECTOR3 vBipedFacing;
    GetSpeedDirection( &vBipedFacing );
    // physical radius
    float fRadius = GetPhysicsRadius();

    // if we have not reached position , we will move on.
    if(bReachPos == false && GetSpeed() != 0.f)
    {
        // get origin
        D3DXVECTOR3 orig = m_vPos;
        float fSensorHeight = GetPhysicsHeight()*SENSOR_HEIGHT_RATIO;
        orig.y += fSensorHeight;
        // compute sensor group 0.
        g_sensorGroups[0].ComputeSensorGroup(orig, vBipedFacing, fRadius);
    }

    bool bCanMove = false; // whether the character can move either directly or sliding along wall.

    // move the character according to its impact forces
    if(g_sensorGroups[0].GetHitWallCount() > 1)
    {
        /** it has hit multiple things, we will not move the object */
        bCanMove = false;
    }
}

```

```

else
{
    if(g_sensorGroups[0].GetHitWallCount() == 1)
    {
        bCanMove = true;
        bSlidingWall = true;

        /**
         * we use the average impact norm to get a tentative point where the biped is most likely to be
         in the next frame.
         * we then cast another ray (group 1) from this tentative point to get another impact point. if
         there is no impact point
         * within the radius of (fRadius+m_fSpeed * dTimeDelta), the object will move using the old
         facing vector.
         * without further processing. Otherwise, from the two impact points(of group 0 and 1), we can
         calculate the wall direction vector,
         * which will be used for sliding wall.
        */
    {
        // we will try sliding the character along the wall.The wall normal is given by impactNorm
        (the surface norm).
        D3DXVECTOR3 vWallNorm = g_sensorGroups[0].GetAvgImpactNorm(); // use only its
projection on the y=0 plane.
        vWallNorm.y = 0;
        D3DXVec3Normalize(&vWallNorm,&vWallNorm);

        D3DXVECTOR3 vTentativeFacing;
        /** we will compute a tentative sliding wall facing(not the final one) as below:
         * vFacing = vWallNorm (X) (vFacing (X) vWallNorm);
         * and get a tentative new position of the character.
        */
        D3DXVec3Cross(&vTentativeFacing, &vBipedFacing, &vWallNorm);
        D3DXVec3Cross(&vTentativeFacing, &vWallNorm, &vTentativeFacing);
        D3DXVec3Normalize(&vTentativeFacing,&vTentativeFacing); // just make it valid
        D3DXVec3Scale( &vTentativeFacing, &vTentativeFacing, PENETRATION_DISTANCE );
        vTentativeFacing.y=0;
        D3DXVECTOR3 vHitRayOrig = g_sensorGroups[0].m_vOrig + vTentativeFacing;
        D3DXVECTOR3 vHitRayDir = g_sensorGroups[0].GetHitRaySensor().vDir;
        float fSensorRange = fRadius+float(m_fSpeed * dTimeDelta);
        // compute sensor group 1.
        g_sensorGroups[1].ComputeSensorGroup(vHitRayOrig, vHitRayDir, fSensorRange, 1);
    }
    bool bMoveAlongOldFacing = false;
    if(g_sensorGroups[1].HasHitAnything())
    {
        // check to see if the object needs to follow the wall.
        D3DXVECTOR3 vWallDir =
(D3DXVECTOR3&)(g_sensorGroups[1].GetHitRaySensor().impactPoint -
g_sensorGroups[0].GetHitRaySensor().impactPoint);
        vWallDir.y=0;
        // g_sensorGroups[1].GetHitRaySensor().fDist;
        if(vWallDir != D3DXVECTOR3(0,0,0))
        {
            D3DXVec3Normalize(&vWallDir,&vWallDir);

            D3DXVECTOR3 tmp1,tmp2;
            D3DXVec3Cross(&tmp1, &g_sensorGroups[1].GetHitRaySensor().vDir, &vWallDir);
            D3DXVec3Cross(&tmp2, &vBipedFacing, &vWallDir);
            if(tmp1.y * tmp2.y > 0)

```

```

    {
        // if the object is walking into the wall, we will slide along the wall.
        float speedScale = D3DXVec3Dot(&vBipedFacing, &vWallDir);
        D3DXVECTOR3 vFacing;
        D3DXVec3Scale( &vFacing, &vWallDir, float( m_fSpeed *
dTimeDelta*speedScale ) );
        D3DXVec3Add( & vMovePos, & vFacing, & m_vPos );
    }
    else
        bMoveAlongOldFacing = true;
}
else
    bMoveAlongOldFacing = true;

if(bMoveAlongOldFacing){
    // maintain the direction.
    D3DXVECTOR3 vFacing;
    D3DXVec3Scale( &vFacing, &vBipedFacing, (float)(m_fSpeed * dTimeDelta));
    D3DXVec3Add( & vMovePos, & vFacing, & m_vPos );
}

if(vMovePos != m_vPos){
{
    D3DXVECTOR3 vOrig = g_sensorGroups[0].m_vOrig + (vMovePos - m_vPos);
    // compute sensor group 2.
    g_sensorGroups[2].ComputeSensorGroup(vOrig, vBipedFacing, fRadius);
}
if(g_sensorGroups[2].HasHitAnything()){
    if(!bMoveAlongOldFacing) && g_sensorGroups[2].CompareWith(g_sensorGroups[0]))
    {
        // if the biped has reached a corner, stop it.
        vMovePos = m_vPos;
    }else {
        float fReboundDistance = (fRadius - g_sensorGroups[2].GetHitRaySensor().fDist -
PENETRATION_DISTANCE);
        if(fReboundDistance >= 0){
            D3DXVECTOR3 vFacing;
            D3DXVec3Scale( &vFacing, &g_sensorGroups[2].GetHitRaySensor().vDir, -
fReboundDistance);
            D3DXVec3Add( & vMovePos, & vFacing, & vMovePos );
        }
    }
}
}else {
    // if nothing is hit, simply move the character forward
    bCanMove = true;
    D3DXVECTOR3 vFacing;
    D3DXVec3Scale( &vFacing, &vBipedFacing, float( m_fSpeed * dTimeDelta ) );
    D3DXVec3Add( & vMovePos, & vFacing, & m_vPos );
}
}

/***
* check to see if the character is in the air, or is climbing up stairs.
* if so we will see the bReachPos to false, and allow the character to smoothly fall down or fly up.
*/

```

```

D3DXVECTOR3 orig = vMovePos;
orig.y += GetPhysicsHeight();
NxRay ray((const NxVec3&)orig, NxVec3(0,-1.f,0));
NxRaycastHit hit;

// Get the closest shape
float dist;
NxShape* closestShape = CGlobals::GetPhysicsWorld()->GetScene()->raycastClosestShape(ray,
NX_ALL_SHAPES, hit, 0xffffffff, 10*OBJ_UNIT);
if (closestShape)
{
    dist = hit.distance;
}
else
{
    dist = MAX_DIST*OBJ_UNIT; // infinitely large
}

// set the object height to the higher of the two.
float fTerrainHeight = CGlobals::GetGlobalTerrain()->GetElevation(vMovePos.x, vMovePos.z);
float fPhysicsHeight = orig.y - dist;

if(fTerrainHeight > (m_vPos.y + GetPhysicsHeight())){
    bIgnoreTerrain = true;
    // the biped has gone into a terrain hole, we will subject the object to the physics object only.
    vMovePos.y = fPhysicsHeight;
    bUseGlobalTerrainNorm = false;
}else{
    bIgnoreTerrain = false;

    /// take terrain in to account, we will adopt the higher of the two as the character's next height.
    if(fPhysicsHeight > fTerrainHeight){
        // physics object is over the terrain objects.
        vMovePos.y = fPhysicsHeight;
        bUseGlobalTerrainNorm = false;
    }else{
        // the terrain is over the physics objects.
        vMovePos.y = fTerrainHeight;
        if(m_vPos.y < (vMovePos.y + GetPhysicsHeight()) )
            /// if the object is not too far from the ground, it will subject to terrain surface norm.
            bUseGlobalTerrainNorm = true;
        else
            /// if the object is well above the ground(in air), it will not snap to terrain surface norm.
            bUseGlobalTerrainNorm = false;
    }
}

if(vMovePos.y < LOWEST_WORLD)
    vMovePos.y = LOWEST_WORLD;

// move the object to the new location in the x,z plane
m_vPos.x = vMovePos.x;
m_vPos.z = vMovePos.z;
/// animate the character vertically according to gravity.
/// implement smooth fall down and jump up.
if(m_vPos.y > vMovePos.y)
{
    if(m_fSpeedVertical == 0.f)
        m_fSpeedVertical = -GetAbsoluteSpeed();
}

```

```

float fLastSpeedVertical = m_fSpeedVertical;
m_fSpeedVertical -= fGravity*(float)dTimeDelta;
float dY = (float)dTimeDelta*(m_fSpeedVertical+fLastSpeedVertical)/2.f;
m_vPos.y += dY;

if(m_fSpeedVertical == 0.f)
    m_fSpeedVertical = EPSILON;

if(m_vPos.y <= vMovePos.y){
    m_fSpeedVertical = 0.f;
    m_vPos.y = vMovePos.y;
    /* end jumping, if the biped is near the ground and has a downward vertical speed.*/
{
    CBipedStateManager* pCharState = GetBipedStateManager();
    if(pCharState)
        pCharState->AddAction(CBipedStateManager::S_JUMP_END);
}
}else if (m_vPos.y > (vMovePos.y+GetPhysicsHeight()*0.25f) && m_fSpeedVertical<0){
    /* fall down, if the biped is well above the ground and has a downward vertical speed.*/
{
    CBipedStateManager* pCharState = GetBipedStateManager();
    if(pCharState)
        pCharState->AddAction(CBipedStateManager::S_FALLDOWN);
}
}
}else if(m_vPos.y < vMovePos.y){
    bool bIsJumpingUp = false;
    float fClimbUpSpeed = GetAbsoluteSpeed();
    if(fClimbUpSpeed < CLIMBUP_SPEED)
        fClimbUpSpeed = CLIMBUP_SPEED;
    if(m_fSpeedVertical <= fClimbUpSpeed )
        m_fSpeedVertical = fClimbUpSpeed;
    else
        bIsJumpingUp = true;

    float fLastSpeedVertical = m_fSpeedVertical;
    m_fSpeedVertical -= fGravity*(float)dTimeDelta;
    float dY = (float)dTimeDelta*(m_fSpeedVertical+fLastSpeedVertical)/2.f;
    m_vPos.y += dY;

    if(bIgnoreTerrain){
        if(m_vPos.y >= vMovePos.y)
            m_vPos.y = vMovePos.y;
    }else {
        if(m_vPos.y<vMovePos.y || (!bIsJumpingUp))
            m_vPos.y = vMovePos.y;
    }
    if(!bIsJumpingUp){
        m_fSpeedVertical = 0.f;
{
        CBipedStateManager* pCharState = GetBipedStateManager();
        if(pCharState)
            pCharState->AddAction(CBipedStateManager::S_JUMP_END);
}
    }
}
}else{
    /* if the character is on the ground, make sure that it has a non-negative speed*/
    if(m_fSpeedVertical > 0.f)

```

```

{
    float fLastSpeedVertical = m_fSpeedVertical;
    m_fSpeedVertical -= fGravity*(float)dTimeDelta;
    float dY = (float)dTimeDelta*(m_fSpeedVertical+fLastSpeedVertical)/2.f;
    m_vPos.y += dY;
}
else
    m_fSpeedVertical = 0.f;
}

// Compute the norm (orientation) of the biped, and smoothly transform to it.
{
    D3DXVECTOR3 vNorm;
    if(bUseGlobalTerrainNorm) {
        // get the global terrain norm
        vNorm = GetNormal();
        // this normal value simulate a real biped on a slope
        if((1.f-vNorm.y) > EPSILON ){
            D3DXVECTOR3 vAxis;
            D3DXVec3Normalize(&vAxis, D3DXVec3Cross(&vAxis, &D3DXVECTOR3(0,1,0), &vNorm));
            D3DXVECTOR3 vFacing;
            GetSpeedDirection(&vFacing);
            float fFactor = D3DXVec3Dot(&vAxis, &vFacing);
            if(fabs(fFactor) < EPSILON)
                fFactor = 1;
            else
                fFactor = sqrt(1-fFactor*fFactor);

            D3DXMATRIX mx;
            D3DXMatrixRotationAxis(&mx, &vAxis, acos(vNorm.y)*fFactor);
            D3DXVec3TransformCoord(&vNorm, &D3DXVECTOR3(0,1,0), &mx);
        }
    }else {
        // For physics meshes. the norm is always (0,1,0)
        vNorm = D3DXVECTOR3(0,1,0);
    }
    CMath::SmoothMoveVec3(&m_vNorm, vNorm, m_vNorm, (SPEED_NORM_TURN*dTimeDelta), 0);
}
if(pIsSlidingWall){
    *pIsSlidingWall = bSlidingWall;
}
return bReachPos;
}

```

8.3 Summary and Outlook

Physics can be very complex in a game engine and there is rarely a single framework of physics which can be applied throughout a game engine. This is because, in computer game engines, some physics are physically simulated, some are motion blended, while others may be triggered by special objects in the scene. Both the physics model representation and simulation algorithms can differ greatly for different types of objects in the 3D world. For example, the global terrain, the weather system, the ocean water, the particle system, the indoor buildings, and outdoor meshes may all have different simulation schemes. Sometimes, integrating all the simulation scheme may not be as easy as choosing the appropriate method for each of them. For example, suppose there is a cave inside the heightmap-based global terrain. A game engine can not decide whether the cave physics or the heightmap-based

global terrain physics should be applied at the intersection regions. In such cases, some additional rules need to be specified when object transfers from one physics system to another.

In ParaEngine, we have implemented physics simulation for the global terrain, the indoor buildings, outdoor meshes and some of their combinations. Mobile characters may be subject to one or more of these physics system at a time. For example, a character may swim from the ocean to the land; fall down a hole on the terrain and landed on the floor of an underground drungon, etc.

Chapter 9 Terrain

Terrain is an important part for outdoor games. It is a continuous undulating grid of meshes that may extend infinitely in all directions. In a single camera shot, we may need to render 250000 square meters or more terrain surface without losing details from the observer. This usually involves dynamically rendering a mesh which has very high triangle counts, and painting them with multiple layers of detailed textures. In this chapter, we will examine the technique used in outdoor terrain rendering. In **Chapter 3**, we have shown the major file formats used in terrain rendering, which is very useful in understanding how the terrain works. The terrain module or terrain engine we are going to build in this chapter supports real-time rendering of infinitely large terrain surface, with dynamic multi-texturing and geo-morphing. It also supports terrain holes, through which a game world could be extended to the underground, such as building caves and tunnels.

9.1 Foundation

9.1.1 Understanding Terrain Data

Understanding the terrain data and the ability of a modern personal computer is very useful to the design of a terrain engine. Recall in **chapter 3**, the terrain in a game world is partitioned in to many equal sized squares as shown in Figure 3.3. Each square is called a terrain tile, and is specified by a 2D coordinate. Each terrain tile references an elevation file and a texture mask file that specifies how this terrain tile should be painted, etc. An elevation file is just a two dimensional array `heightmap[nWidth][nHeight]`, where `heightmap[x][y]` contains the terrain height value at $(x * \text{VertexSpacing}, y * \text{VertexSpacing})$. During terrain rendering, the heightmap are used to build the triangles mesh, as shown in Figure 9.1.

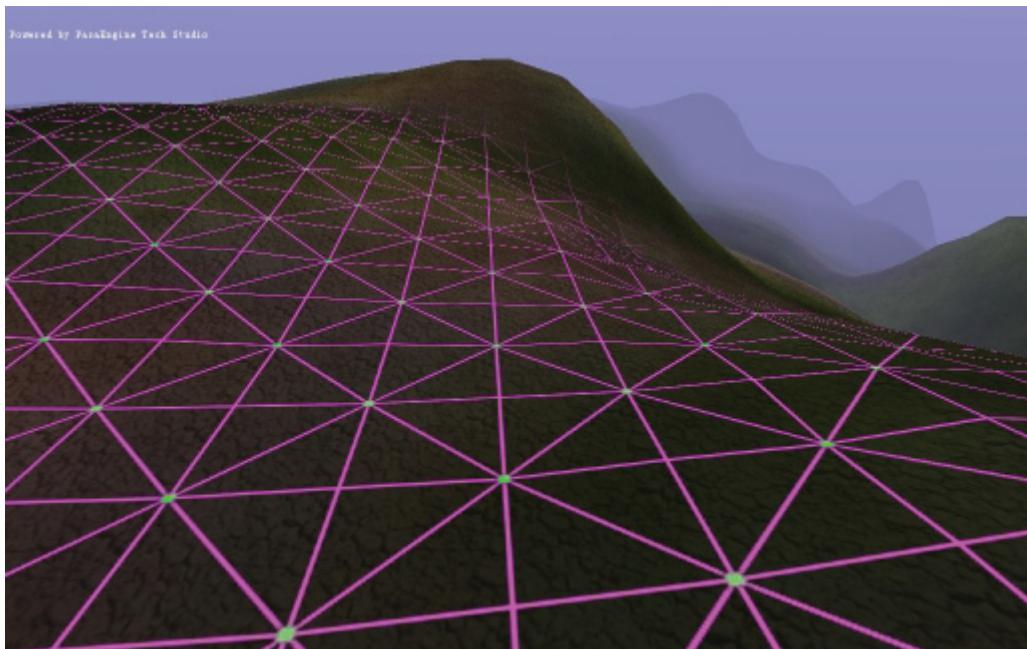


Figure 9.1 Terrain Grid

VertexSpacing is the distance between two adjacent points. Now suppose that VertexSpacing is 1 meter. Then in order to rendering terrain of size 500*500 square meters with brutal force, 250000 triangles should be built and rendered at real time. This is barely affordable for a modern GPU, especially when there are many other scene objects to be rendered in the same frame. Hence we need some algorithm to build the terrain mesh with fewer triangles. From Figure 9.1, we see that the farther away the terrain, the more dense the triangles. In the farthest end, we can barely distinguish the difference between adjacent triangles. On the other hand, notice that some terrain surface is relatively flat with regard to the camera, and we can not distinguish the height difference either. In both cases, we can simplify the terrain mesh by merging triangles. Most popular terrain rendering algorithms are based on this level of detail (LOD) technique. We will see some of them in the next section.

Another large dataset for terrain rendering is texture data which is used for painting the terrain surface. See Figure 3.4 in Chapter 3. First of all, painting the terrain with non-repeated detailed textures is too expensive in terms of memory usage. A 256*256 pixel bitmap can only be used to texture a very small region such as 4*4 square meters. To render a single static frame containing 250000 square meters terrain with just one texture layer, we will need gigabytes of texture data. This is not affordable by current hardware. Moreover, it is difficult for artists to generate such large texture data unless they are taken by satellite photographing. Hence, terrain texturing in computer games is usually implemented as several layers of tiled images blended together. In each texture layer, the same detailed tiled image is repeated across a relatively large region and a very low resolution alpha map is stretched across the entire region to blend the tiled images with the background. See Figure 9.2.

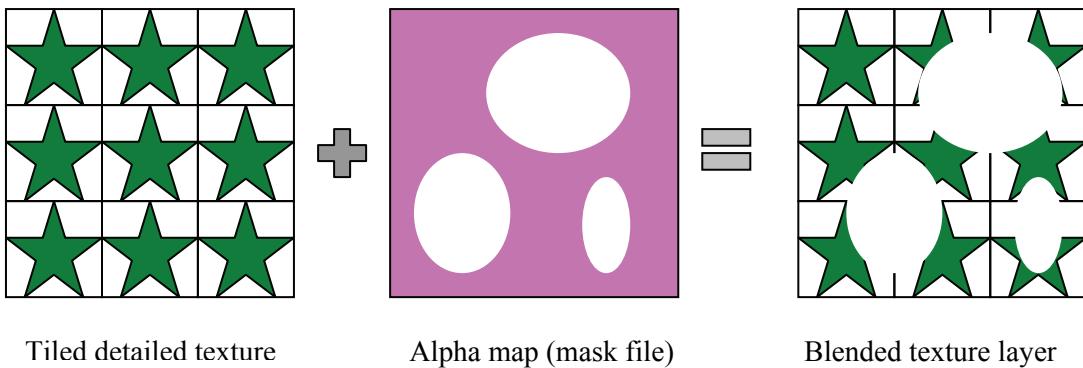


Figure 9.2 Alpha blending of texture layer

Each sub region of a terrain surface may be textured by further blending several alpha blended texture layers. For example, one layer uses a tiled cobble image, another uses tiled grass image, the third uses just 1 pixel black image. Blending these three over a large region using the above multi-texturing technique, we can easily create a grass land with cobble roads and black shadows of props on the land. With tiled images and alpha blending, we need less texture to paint the terrain. However, we still need to provide the alpha map for the entire terrain surface. Fortunately, each pixel of alpha map is 4 or 8 bits and a 128*128 pixel alpha map can be stretched and used for a very large region, such as 60*60 square meters. The texture data for a terrain is thus mainly comprised of alpha maps and a few detailed textures. Alternatively, we call alpha maps, mask files in this book.

9.1.2 Level of Detail Terrain Algorithm

This section provides an overview to several popular terrain rendering algorithms used in game engines. It helps engine programmers to select the proper implementation. The Virtual Terrain Project¹³ provides useful links to published papers and implementations of various terrain rendering algorithms. Popular algorithms used in game terrain rendering are brutal force, Chunked LOD, modified ROAM, GeoMipmapping, which are briefly introduced and compared in this section. We will discuss the hybrid algorithm we used in further details in the next section.

Brutal Force

Brutal force renders the mesh in the view frustum in their original form. Figure 9.3 shows triangles with brutal force terrain rendering. Brutal force rendering is actually very useful in computer games. It is also the algorithm that uses least CPU time. As GPU is getting faster, we can just send the clipped terrain mesh to GPU for rendering with brutal force method. Another special usage of brutal force rendering is that we can use it to render remote terrains which are displayed as silhouette in fog. Brutal force also simplifies the issues of terrain geo-morphing and multi-texturing. It will perhaps be used more often in games as CPU time is getting more precious in computer game engine. Please note that brutal force based terrain usually uses a quad tree for view clipping, so that regions outside the camera view frustum will be clipped quickly. An improved version of brutal force is chunked LOD. Please see below.

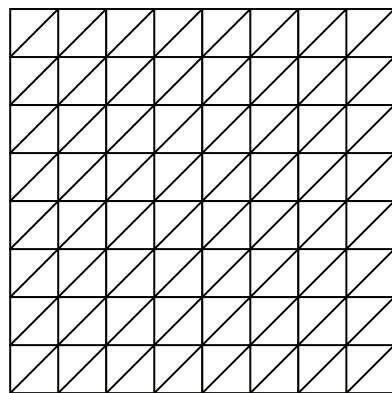


Figure 9.3 Brutal Force Terrain Triangles

Chunked LOD: Chunked level of detail

It is a relatively simple, yet very effective terrain rendering technique based on discrete level of detail. Contrary to other more complicated algorithms, its aim is not to achieve an optimal triangulation, but to maximize triangle throughput, while minimizing CPU usage. It is built on the fact, that today's GPU are so fast that it is usually cheaper to render a bit more triangles than to spend a lot of CPU cycles to drop some of the unnecessary ones. Of course, level of detail management is still required, but at a much coarser level. The basic idea is to

¹³ The Virtual Terrain Project: <http://www.vterrain.org>

apply view dependent LOD management not to single vertices, but to bigger chunks of geometry.

As a preprocessing step, a quad tree of terrain chunks is built. Every node is assigned a chunk, containing the self contained mesh in that quad. The meshes on different levels in the quad tree have different sizes in world space. During rendering, the quad tree nodes are culled against the viewing frustum as usual. The visible nodes are then split recursively, until the chunk's projected error falls below a threshold.

Since the terrain is assembled from separate chunks, there will be cracks in the geometry between different LODs in the hierarchy. In order to minimize these artifacts, Ulrich uses a technique called skirting. Skirting basically means that on the edges of the mesh, there are extra triangles extending downwards, effectively hiding the gaps. See Figure 9.4 for meshes generated by Chunked LOD. In different implementations, the triangle chunk may be constructed differently from the pattern shown in the figure.

Popping is very noticeable with discrete LOD methods, because a lot of the vertices change abruptly when switching to a different level of detail mesh. This disturbing effect can be eliminated by smooth distance based vertex morphing.

Because Chunked LOD uses a quad tree, it is possible to build a huge world using tiled chunks of geometry. It is the most popular terrain rendering technique in today's game engine.

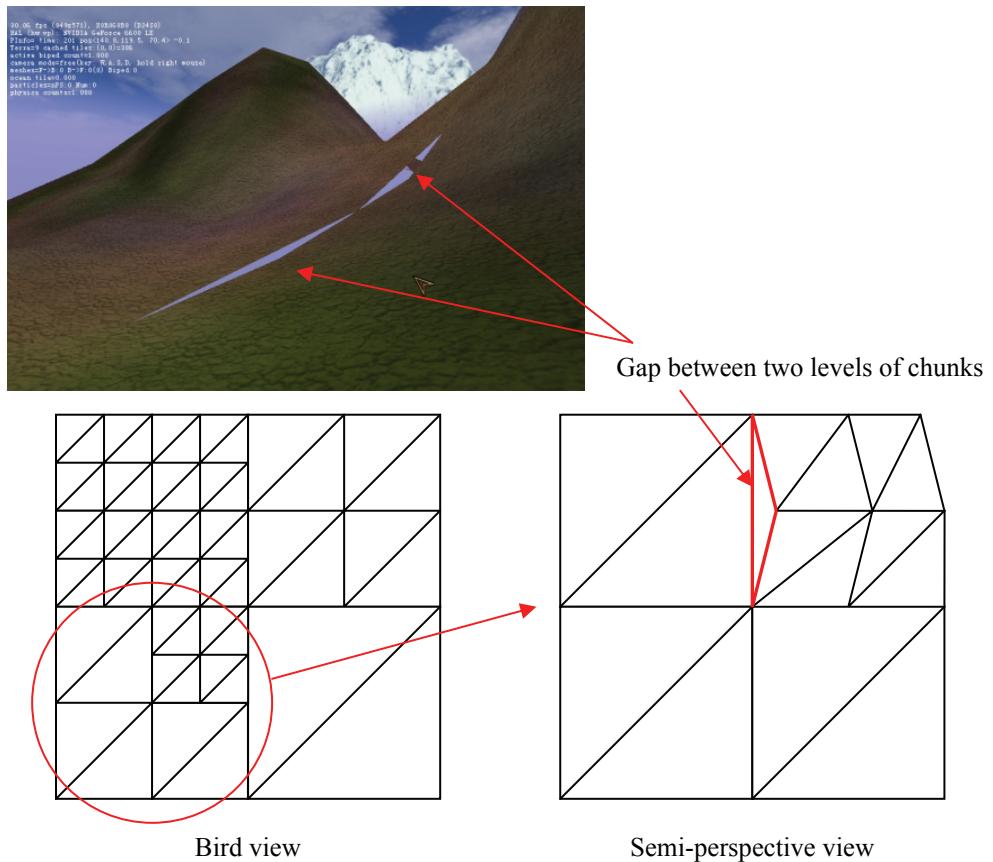


Figure 9.4 Chunked LOD terrain triangulation

Geomipmapping

This algorithm resembles Chunked LOD. It has the analogy in texture mipmapping. Consider the ordinary mipmapping technique for textures. A chain of mips is created for each texture, the first item in the chain is the actual texture of its original size; each succeeding item is the previous item scaled down to half its resolution, until a desired number of items (levels) is reached. When a texture is at certain distance from the camera, the appropriate level in the mipmap chain is chosen and used for rendering instead of the actual high resolution texture. We can apply this concept to three-dimensional meshes. Geomipmapping has the same gap (crack) problem as in chunked LOD and additional triangles are used to repair the cracks. Figure 9.5 shows a possible triangle mesh rendered with geomipmapping.

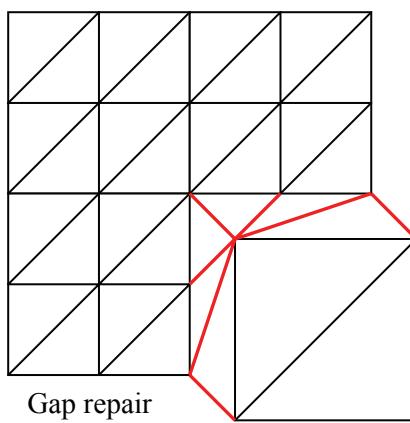


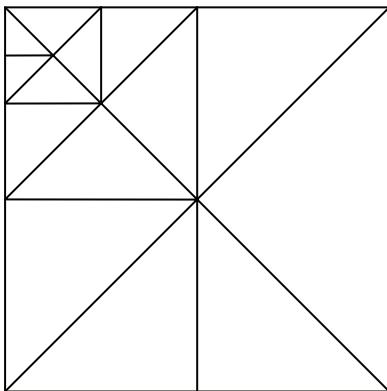
Figure 9.5 Geomipmapping terrain triangulation

ROAM: Real-Time Optimally Adapting Mesh

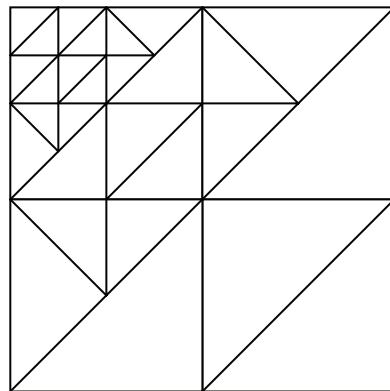
Adapting meshes is a key concept in this algorithm. In case there is no sudden change in the viewpoint, the algorithm is designed to exploit frame to frame coherence. It assumes that the mesh in the next frame will only differ in a few triangles. This means that there is no need to regenerate the whole mesh from scratch every frame.

ROAM maintains two priority queues that drive the split and merge operations. The queues are filled with triangles, sorted by their projected screen space error. In every frame some triangles are split, bringing in more detail, and others are merged into coarser triangles. Since the triangles are progressively refined, it is possible to end the refinement at any time, enabling the application to maintain a stable frame rate. A traditional ROAM algorithm using binary triangle tree will produce mesh that looks like the left one in Figure 9.6, other modified version of ROAM using quad triangle tree may produce triangles that looks like the right one in the figure.

The original ROAM algorithm produces the optimal (minimum triangle count) mesh, but it is very CPU limited. Many game engines rather use a more CPU friendly version of ROAM which uses a quad tree and rebuilds the mesh every frame without using priority queues.



ROAM with binary tree



ROAM with quad tree

Figure 9.6 ROAM terrain triangulation

9.1.3 The Hybrid Terrain Algorithm

Start from Requirement

The choice of terrain algorithm depends on the requirement of the game engine. Our requirement of the terrain engine is given below.

- Tiled terrains: tiled terrain means the game engine need to support infinitely large terrain. This requires that we can efficiently repair cracks on the terrain tile borders.
- Geomorphing: We can change the shape of the terrain at real time. This requires that terrain height map preprocessing should be fast.
- Multi-texturing: This requires that assigning appropriate texture coordinates to triangles should be fast.
- Ray picking: we should be able to query ray-terrain intersections fast.
- Holes: we should be able to dynamically dig holes on terrain surface, so that we can have caves and tunnels. See Figure 9.7.

Starting from these requirements, we use a hybrid approach of chunked LOD and ROAM. It uses a quad tree to store terrain mesh at different level of details, and for every frame, it breadth-first traverses the quad tree to built triangles similar to the one generated by ROAM. See Figure 9.6 (right).

The advantages of the hybrid approach are:

- It is almost as fast as chunked LOD.
- Since the triangles are progressively refined, it is possible to end the refinement at any time. For example, we can specify an upper limit to the number of triangles for each frame, so that the terrain can always be rendered using triangles less than the specified value.
- It satisfied all the game requirements defined above.

There is an open source project called Demeter Terrain Visualization Library, which uses this method. The implementation in this book is based on it and improved in several places.

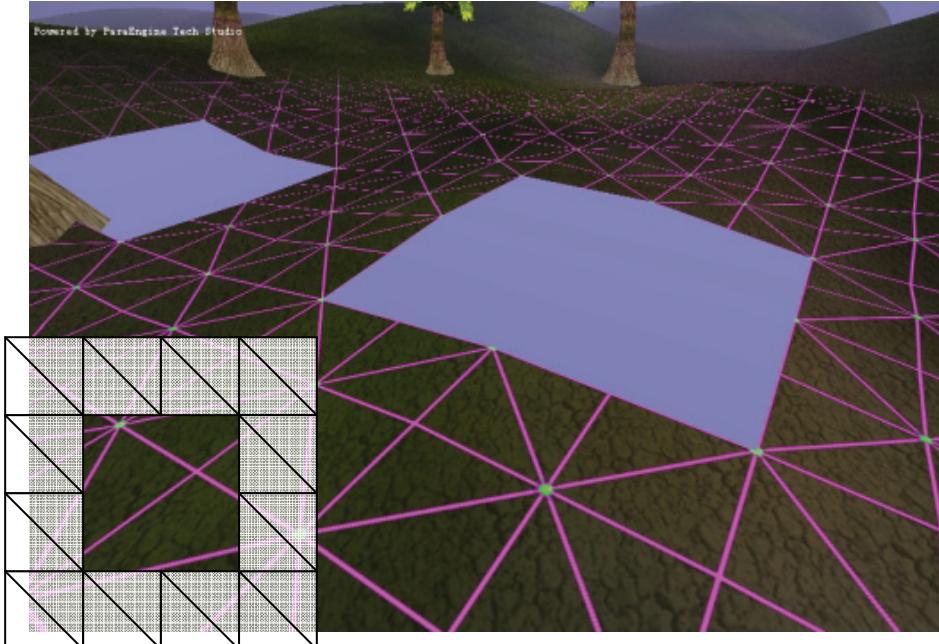


Figure 9.7 Holes on terrain surface

Implementation

We use a terrain object to represent a latticed terrain tile. A terrain object may be adjacent to eight other terrain objects. We will first consider a single terrain object. See Figure 9.8. CGlobalTerrain is the entire (inifinitely large) terrain; Terrain is a single terrain tile object.

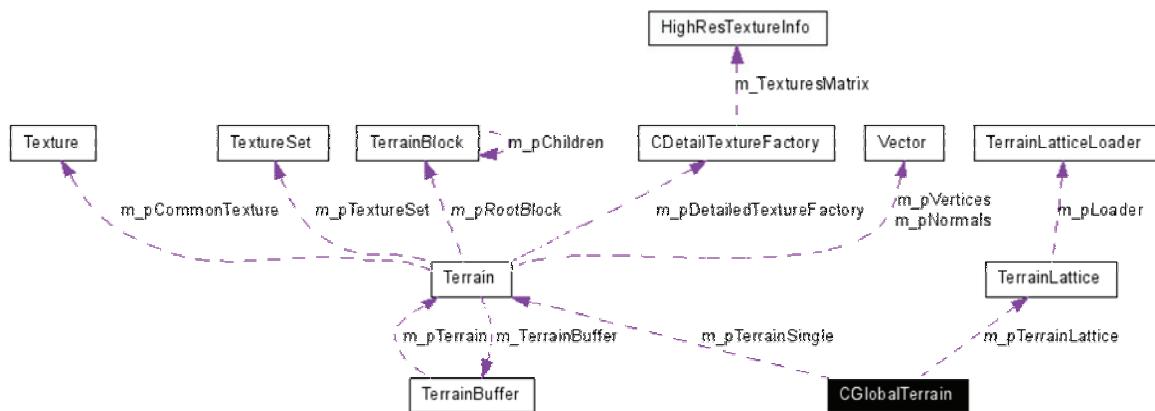


Figure 9.8 Collaboration diagram for Terrain Object

The terrain object can be loaded from the Terrain Config File (See **Chapter 3**). It uses a array (*m_pVertices*) of vectors to store the elevation (heightmap) data. The data for each LOD is pre-computed and stored in a quad tree data structure called *m_pRootBlock*. Each LOD is

also called a terrain block. Each terrain block contains the pre-computed the axis aligned bounding box (only minimum and maximum height is needed) and some other information shown below.

```
/** it represents the quad tree node of LOD terrain data*/
class TerrainBlock
{
    /** repair cracks in this block and child blocks. This is a recursive function.*/
    void CalculateGeometry(Terrain * pTerrain);
    /** generate triangles for rendering in higher LOD. This is a recursive function.*/
    void Tessellate(const double *pMatrixModelview, const double *pMatrixProjection, const int
    *pViewport, TriangleStrip * pTriangleStrips, unsigned int *pCountStrips, Terrain * pTerrain);
    /** repair cracks in this block and child blocks. This is a recursive function.*/
    void RepairCracks(Terrain * pTerrain, TriangleFan* pTriangleFans, unsigned int* pCountFans);

    bool IsActive();
    int GetStride();
    void EnableStrip(bool bEnabled);
    int GetHomeIndex();
    void VertexChanged(Terrain * pTerrain);
    void VertexChanged(Terrain * pTerrain, int index1);
    void VertexChanged(Terrain * pTerrain, int index1, int index2);
    void IntersectRay(const Ray & ray, Vector & intersectionPoint, float &lowestDistance, const
    Terrain * pTerrain);
    /** update holes in the terrain. This will set the m_bIsHold attribute of the terrain block. This function
    will recursively call itself. Call this function when the terrain block has already been built.*/
    void UpdateHoles(Terrain * pTerrain);

private:
    /// quad tree data structure. 4 valid pointers for non-leaf nodes
    TerrainBlock **m_pChildren;
    int m_HomeIndex;
    short int m_Stride;
    /// whether this block is a hole.
    bool m_bIsHole:1;
    bool m_bIsActive:1;
    bool m_bChildrenActive:1;
    TriangleStrip *m_pTriangleStrip;
    float m_MinElevation, m_MaxElevation; /// bounding box
#if _USE_RAYTRACING_SUPPORT_ > 0
    Triangle *m_pTriangles;
#endif
    friend class Terrain;
    ...// some function omitted.
};
```

Figure 9.9 shows the bounding box of quad trees. The terrain height value at the four vertices of each terrain block will be used to build simplified triangles at that LOD. For performance reasons, we also pre-compute, in each block, the index of the upper left vertex in the terrain's elevation array (m_pVertices). This index is called home index to that terrain block.

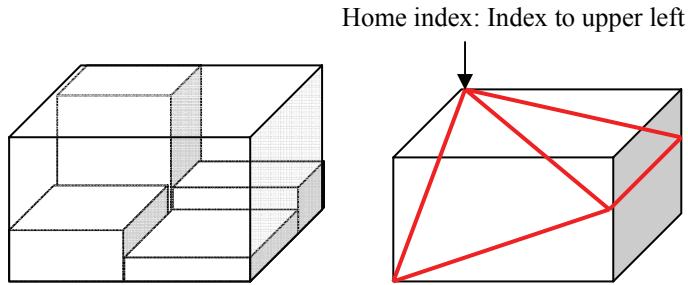


Figure 9.9 Bounding Box and Triangles in Terrain quad tree

The basic algorithm works as follows. It consists of the terrain initialization and rendering process.

```

Initialization{
    Parse Terrain Config File

    Load elevation data to Terrain Object{
        Initialize Terrain::m_pVertices as an array of vectors
        Precompute quad tree (m_pRootBlock){
            m_pVertexStatus = initialize vertex status, which is an array of bits with the same dimension
                of the heightfield. This is used for marking vertices during data processing.
            m_pRootBlock = Recursively create Terrain Blocks and store them in a quad tree
            // recursively compute the bounding volume and home index for all blocks in a quad tree.
            m_pRootBlock->CalculateGeometry();
        }
    }

    Load hole map to Terrain Object as an array of Booleans.
    // Recursively update the quad tree to set the m_bIsHole attribute for each terrain block.
    m_pRootBlock->UpdateHoles()

    Load Texture set
    Load texture masks
}

For each frame during rendering{
    If(camera has not moved since last frame){ render using old triangle buffer }
    // the tessellation function will build a list of triangles
    Tessellate(){
        // recursively tessellate the terrain and generate triangles at the proper LOD.
        m_pRootBlock->Tessellate()
        breadth-first traversing quad tree terrain blocks{
            if(the block is not a hole){
                if (block's bounding box is inside the view frustum){
                    if(triangle count does not exceed the upper limit){
                        If(block is too big to be textured or ...){
                            Inactivate the block, traverse child blocks
                        }else{
                            Project the block to screen space
                            If(the block is too small to be noticed){
                                add (simplified) triangles in this block to the list, do not traverse child blocks
                            }else{
                                Inactivate the block, traverse child blocks
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
        if(block is leaf node whose child need to be traversed){
            add triangles in this leaf block to the list
        }
    }else{
        If(block is too big to be textured or ...){
            Inactivate the block, traverse child blocks
        }else{
            add (simplified) triangles in this block to the list, do not traverse child blocks
        }
    }
}else{
    Inactivate the block, do not traverse child blocks
}
}else{
    Inactivate the block, do not traverse child blocks
}
}
}

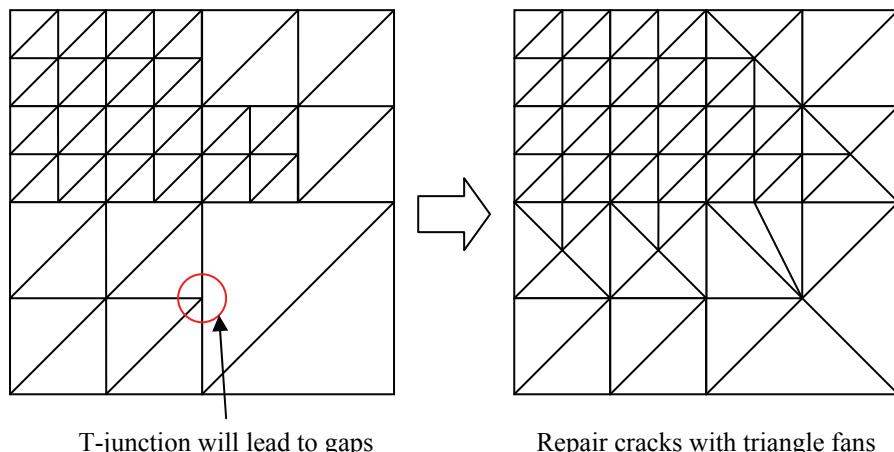
RepairCracks(){
    Repair the cracks (gap) between two adjacent LOD triangle soups.
}
RebuildRenderBuffer(){
    Sort triangles by texture
}

Render(){
    Render each group of triangles with multi-texturing
}
}

```

During each frame, we need to tessellate the terrain according to current camera viewpoint, repair cracks between two adjacent level-of-details, building the triangle buffer and render the triangles with multi-texturing.

In the tessellation step, we will produce triangles in the pattern in Figure 9.3. In the repair step, we can effectively remove T-shaped junctions by changing the triangle topologies in the lower level of detail block as shown in Figure 9.10. Alternatively, we can insert additional triangles at the crack as illustrated by Figure 9.4 and Figure 9.5.



T-junction will lead to gaps

Repair cracks with triangle fans

Figure 9.10 Repair cracks in terrain triangles

After sorting triangles by their texture, we can render them with multi-texturing. The details will be given next.

9.1.4 Painting the Terrain

In previous sections, we have shown that terrain surface is painted by alpha blending several texture layers. Most GPU can do 4 to 8 texture blending in a single pass. Some layer may have special meanings, such as shadow map (light map). Some layer may be blended differently, such as using bump mapping. They are all up to the engine designer's choice for the way textures are blended (i.e. blending order, blending weight and method). We will see here our general implementation in ParaEngine.

We define the following texture layers, all of which are optional during terrain rendering.

name	symbol	TexCord	Description
Base Layer	B	u1,v1	Low resolution non-repeated texture, it provides the basic hue of the terrain.
Dirt Layer	D	u2,v2	A tiled detailed or bump texture blended on the base layer to make it look coarser.
The i^{th} Alpha layer	D_i	u2, v2	Tiled detailed texture (D_i) which is blended using an alpha map (A_i).
	A_i	u1,v1	

The final pixel color is given by the following formula.

$$\begin{cases} T_0 = \{B.rgb \times D.rgb, B.a \times D.a\} \\ T_i = \{D_i.rgb, D_i.a \times A_i.a\} \\ C_0 = T_0.rgb \\ C_i = C_{i-1} \times (1 - T_i.a) + T_i.rgb \times T_i.a \end{cases} \quad \text{or} \quad \begin{cases} C_0 = B.rgb \times D.rgb \\ C_i = C_{i-1} \times (1 - D_i.a \times A_i.a) + D_i.rgb \times (D_i.a \times A_i.a) \end{cases}$$

C_n denotes the final pixel color after blending n layers. The left one can be easily matched to DirectX's fixed rendering pipeline; the right one can be easily implemented in a pixel shader. They are functional equivalent. If pixel shader is supported, the following alternative blending method may be much easier to understand and implement.

$$C_n = (B.rgb \times D.rgb \times W_0 + \sum_{i=1}^n D_i.rgb \times (D_i.a \times A_i.a) \times W_i), \text{ where } \sum_{i=0}^n W_i = 1$$

[x].rgb means the RGB color component of the pixel, [x].a means its alpha component.

W is the blending weight, one can use $W_i = 1/n$ or any other weight distribution. The two methods described above are not equivalent. In ParaEngine, we use the first method.

For each terrain vertex, we will supply two texture coordinates (u1, v1) and (u1, v2). (u1, v1) is the low resolution texture coordinate for B and A_i . Images are stretched in this texture coordinate system. (u2, v2) is the high resolution texture coordinate for D and D_i . Images are not stretched in this texture coordinate system. The two coordinate systems usually relate to

each other by an integer scale factor. In our implementation the scale factor is 16 by default. In other words:

$$\begin{cases} u2 = u1 \times 16 \\ v2 = v1 \times 16 \end{cases}$$

To minimize seams introduced by stretching, we need to enable texture clamping for B and A_i ; and enable texture wrapping for D and D_i , because they are tiled detailed textures.

Figure 9.11 are sample images for B , D and D_i .

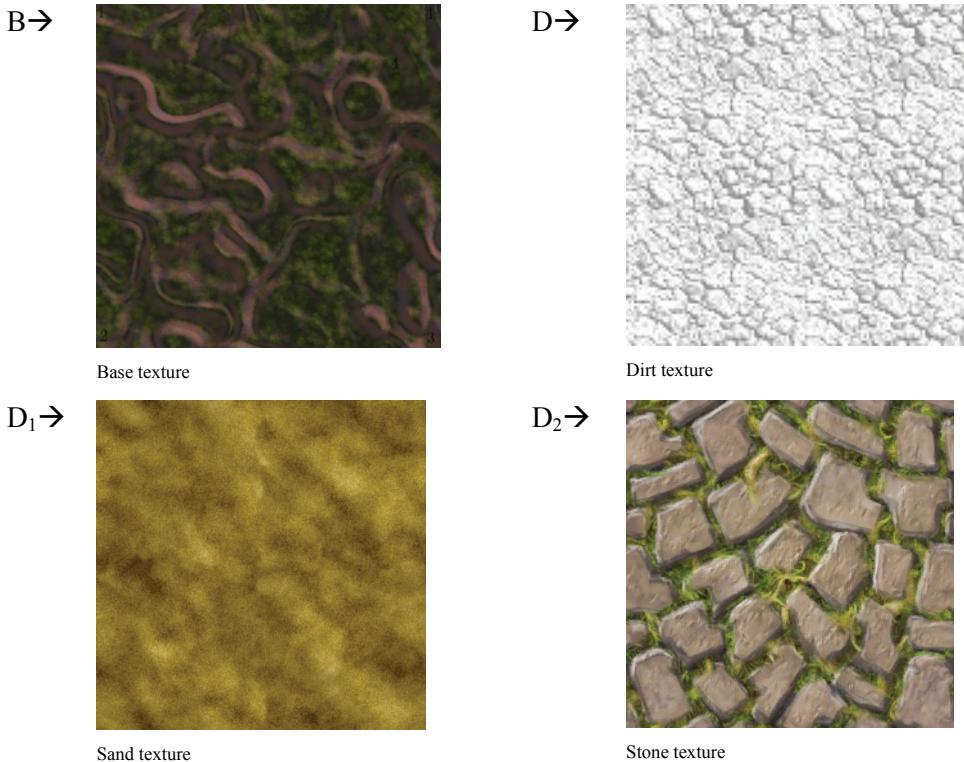


Figure 9.11 Terrain texture samples

9.1.5 Miscellaneous Functions

There are a few functions that we have not covered. They are terrain holes, terrain morphing and dynamic painting. They are not difficult to implement, we will talk about them briefly.

Terrain holes

We can create an array of Booleans called `hole_map` to represent whether there is a hole on the terrain block. Also, when pre-computing the terrain quad tree, we will mark the attribute bit for any block that is a hole. During terrain tessellation phase, blocks with the attribute bit set will not be rendered; neither will its child blocks be rendered. During ray-picking on the terrain surface, whenever an intersection is found, the hit point will be further checked with the `hole_map` to see if it falls into a hole.

Terrain Geo-Morphing

Terrain Geo-morphing means dynamically changing the height field of the terrain. The first step is to modify the height field array. And then update (recompute) the affected quad-tree. Usually only a sub region of the terrain height field is modified, and we do not need to recompute the entire terrain.

Dynamic painting

Dynamic painting refers to editing the detailed textures (D_i and A_i) on terrain surface. This is how we paint the mountain top with snow and stone road in front of a house, etc, in a game. What we paint on is actually the alpha map (mask), i.e., the A_i textures. In ParaEngine, a single terrain surface is covered by a grid (default is 8x8 in ParaEngine) of alpha maps, each of which is an 8-bit texture of 256*256 or less pixels in size. The alpha map is made small so that updating them is fast and that we do not waste too much texture memory when painting on a small region. If the paint brush is at the boundary of two or three alpha maps, they must all be updated.

9.1.6 Latticed Terrain

So far, we have only studied the implementation for a single square terrain. We can use the method described above to construct terrain with height field as large as 4096*4096. This is enough for most game applications. However, loading a terrain that large is time consuming and it makes poor memory usage. Moreover, we want our terrain to be infinitely large. The solution is to use latticed terrain, which is a grid of single terrain objects.

We can load terrain tiles on demand and render them separately. Latticed terrain is a grid of single terrain objects. The only problem to solve is that how we deal with cracks between two adjacent terrains in the lattice. Here we will suppose that the terrain height field is $(2^n+1)^2$ and that the height fields of any adjacent terrains share the same boundary height value. Hence, the cracks can only result from the tessellation phase when two adjacent terrains use two different LOD blocks at the boundary. See Figure 9.12.

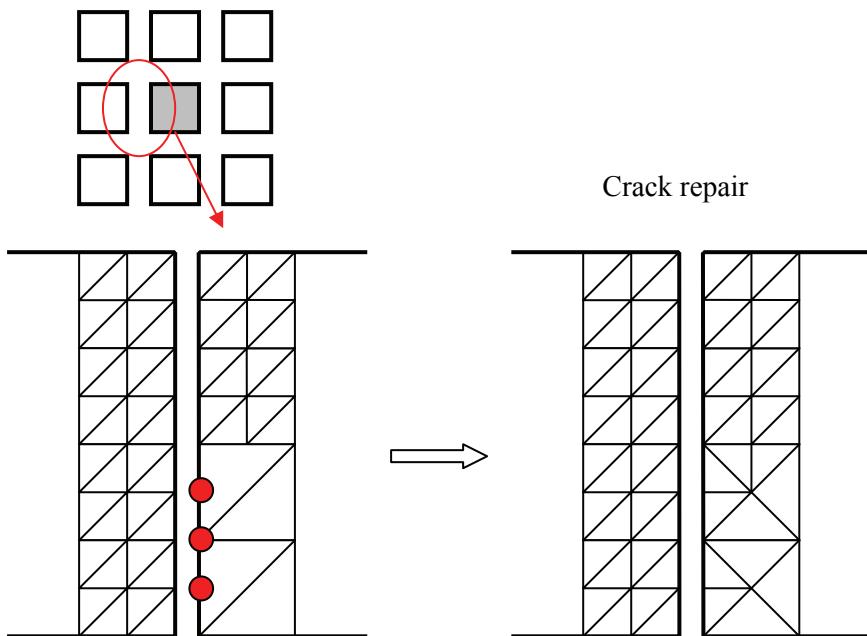


Figure 9.12 Crack repair for latticed terrain

The solution is to insert another step between the tessellation phase and the crack repair phase. In this inserted step, each loaded terrain will check its adjacent 8 terrains and mark on its boundary vertex states (the three red dots in Figure 9.12), so that during the crack repair stage of each terrain, it will know how to deal with it by examining the vertex states. The modified algorithm is given below.

```
For each frame during rendering{
    If(camera has moved since last frame){
        For each potentially visible terrain{
            terrain→Tessellate(){}
        }
        For each potentially visible terrain{
            For each terrain_neighbour of the 8 adjacent terrain {
                terrain→UpdateByNeighbour(terrain_neighbour);
                terrain_neighbour→UpdateByNeighbour(terrain);
            }
        }
        For each potentially visible terrain{
            terrain→RepairCracks(){}
            terrain→ RebuildRenderBuffer){}
        }
    }
    For each potentially visible terrain{
        terrain→Render(){}
    }
}

Terrain::UpdateByNeighbour(terrain_Neighbour){
    Mark all boundary vertices in this terrain, which are present in the terrain_Neighbour.
}
```

9.2 Architecture and Code

The architecture is already introduced in above section. We will show the actual code for the main algorithm discussed in this chapter.

9.2.1 Tessellation Code

```
void TerrainBlock::Tessellate(const double *pMatModelView, const double *pMatProjection, const int *pViewport, unsigned int *pCountStrips, Terrain * pTerrain)
{
    queue<TerrainBlock*> queueBlocks;
    queueBlocks.push((TerrainBlock*)this);

    /// breadth first traversing the quad tree
    while(!queueBlocks.empty()) {
        TerrainBlock* pBlock = queueBlocks.front();
        queueBlocks.pop();
        if(pBlock->Tessellate_NonRecursive(pMatModelView, pMatProjection, pViewport, pCountStrips,
pTerrain)) {
            queueBlocks.push(pBlock->m_pChildren[0]);
            queueBlocks.push(pBlock->m_pChildren[1]);
            queueBlocks.push(pBlock->m_pChildren[2]);
            queueBlocks.push(pBlock->m_pChildren[3]);
        }
    }
}

bool TerrainBlock::Tessellate_NonRecursive(const double *pMatModelView, const double *pMatProjection, const int *pViewport, unsigned int *pCountStrips, Terrain * pTerrain)
{
    bool bProcessChild = false;
    if(!m_bIsHole){
        Box boundingBox;
        float width = m_Stride * pTerrain->GetVertexSpacing();
        boundingBox.m_Min.x = pTerrain->m_pVertices[m_HomeIndex].x;
        boundingBox.m_Min.y = pTerrain->m_pVertices[m_HomeIndex].y;
        boundingBox.m_Min.z = m_MinElevation;
        boundingBox.m_Max.x = boundingBox.m_Min.x + width;
        boundingBox.m_Max.y = boundingBox.m_Min.y + width;
        boundingBox.m_Max.z = m_MaxElevation;

        if (pTerrain->CuboidInFrustum(boundingBox)){
            if(*pCountStrips < pTerrain->m_MaxNumberOfPrimitives) {
                if (pTerrain->m_MaximumVisibleBlockSize < m_Stride ||
                    pTerrain->CuboidInDetailedRadius(boundingBox) ){
                    bProcessChild = true;
                    m_bIsActive = false;
                    m_bChildrenActive = true;
                }else {
                    // Check screen coordinates of center of each face of bounding box
                    D3DXVECTOR3 screenTop, screenBottom;
                    float halfWidth = (boundingBox.m_Max.x - boundingBox.m_Min.x) / 2;
                    // calculate z half way up the BoundingBox
                    float CenterZ = (boundingBox.m_Min.z + boundingBox.m_Max.z) * 0.5f;
                    float screenDistSqure;
                }
            }
        }
    }
}
```

```

float faceX, faceY, faceZ;
// bottom face
faceX = boundingBox.m_Min.x + halfWidth;
faceY = boundingBox.m_Min.y + halfWidth;
faceZ = boundingBox.m_Min.z;
myProject(faceX, faceY, faceZ, pMatModelView, pMatProjection, pViewport,
&screenBottom);
// top face
faceZ = boundingBox.m_Max.z;
myProject(faceX, faceY, faceZ, pMatModelView, pMatProjection, pViewport, &screenTop);

float deltaX, deltaY;//, deltaZ;
deltaX = (float)(screenTop.x - screenBottom.x);
deltaY = (float)(screenTop.y - screenBottom.y);
screenDistSqure = (deltaX * deltaX + deltaY * deltaY);

if (screenDistSqure <= pTerrain->GetDetailThreshold()){
    // This block is simplified, so add its triangles to the list and stop recursing.
    CreateTriangleStrip(pCountStrips, pTerrain);
    m_bIsActive = true;
    m_bChildrenActive = false;

    // set lowest visible height
    if(m_MinElevation < pTerrain->GetLowestVisibleHeight())
        pTerrain->SetLowestVisibleHeight(m_MinElevation);
}else{
    bProcessChild = true;
    m_bIsActive = false;
    m_bChildrenActive = true;
}
/** if it is the leave node that need to be further tessellated, build the triangles directly.*/
if ((mStride == 2) && (m_bChildrenActive)){
    // set lowest visible height
    if(m_MinElevation < pTerrain->GetLowestVisibleHeight())
        pTerrain->SetLowestVisibleHeight(m_MinElevation);

    int offset;
    int nWidthVertices = pTerrain->GetWidthVertices();
    int nLastStripCount = *pCountStrips;

    pTerrain->SetVertexStatus(m_HomeIndex, 1);
    TriangleStrip* tri_strip = pTerrain->GetSafeTriStrip(nLastStripCount);
    tri_strip->m_pVertices[0] = m_HomeIndex;
    offset = m_HomeIndex + nWidthVertices;
    pTerrain->SetVertexStatus(offset, 1);
    tri_strip->m_pVertices[1] = offset;
    offset = m_HomeIndex + 1;
    pTerrain->SetVertexStatus(offset, 1);
    tri_strip->m_pVertices[2] = offset;
    offset += nWidthVertices;
    pTerrain->SetVertexStatus(offset, 1);
    tri_strip->m_pVertices[3] = offset;
    offset = m_HomeIndex + 2;
    pTerrain->SetVertexStatus(offset, 1);
    tri_strip->m_pVertices[4] = offset;
    offset += nWidthVertices;
    pTerrain->SetVertexStatus(offset, 1);
    tri_strip->m_pVertices[5] = offset;
}

```

```

        tri_strip->m_NumberOfVertices = 6;
        tri_strip->m_bEnabled = true;

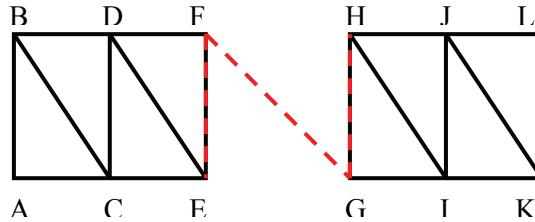
        offset = nWidthVertices + m_HomeIndex;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip = pTerrain->GetSafeTriStrip(nLastStripCount+1);
        tri_strip->m_pVertices[0] = offset;
        offset += nWidthVertices;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip->m_pVertices[1] = offset;
        offset = nWidthVertices + m_HomeIndex + 1;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip->m_pVertices[2] = offset;
        offset += nWidthVertices;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip->m_pVertices[3] = offset;
        offset = nWidthVertices + m_HomeIndex + 2;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip->m_pVertices[4] = offset;
        offset += nWidthVertices;
        pTerrain->SetVertexStatus(offset, 1);
        tri_strip->m_pVertices[5] = offset;
        tri_strip->m_NumberOfVertices = 6;
        tri_strip->m_bEnabled = true;
        tri_strip->textureId = -1;
        *pCountStrips = nLastStripCount + 2;

        m_nTriangleStripIndex = -1;
        m_bIsActive = true;
        m_bChildrenActive = false;
        bProcessChild = false;
    }
} //if(*pCountStrips < pTerrain->m_MaxNumberOfPrimitives)
else{
    if(pTerrain->m_MaximumVisibleBlockSize < m_Stride){
        bProcessChild = true;
        m_bIsActive = false;
        m_bChildrenActive = true;
    }else {
        // This block is simplified, so add its triangles to the list and stop recursing.
        CreateTriangleStrip(pCountStrips, pTerrain);
        m_bIsActive = true;
        m_bChildrenActive = false;
        // set lowest visible height
        if(m_MinElevation < pTerrain->GetLowestVisibleHeight())
            pTerrain->SetLowestVisibleHeight(m_MinElevation);
    }
}
} //else{
m_bIsActive = false;
m_bChildrenActive = false;
}
} //else{
m_bIsActive = false;
m_bChildrenActive = false;
}
}
return bProcessChild;
}

```

9.2.2 Building Triangle Buffer

The following code shows building a triangle list sorted by texture ID. Alternatively, one can use triangle strip with degenerate triangles to render the terrain. Using triangle strip means sending less data to GPU and is faster than using triangle list. Degenerate triangle is triangle that is not visible. It is used to concatenate disjoint triangle strips.



$$\begin{aligned}
 &= ABCDEF + FGGH + GHIJKL \\
 &= .. + DEF + \textcolor{red}{EFF + FFG + FGG + GGH + GHG + GGH} + GHI + ..
 \end{aligned}$$

Figure 9.13 Degenerate Triangles

In Figure 9.13, the triangles in red are degenerate triangles (they share a vertex) and will not be visible. Generally speaking, concatenating triangle strips helps improve batching, and in the case of indexed primitives, there are fewer wasted transformations.

```

/** this is the most time consuming task compared with tessellation and repair crack */
void TerrainBuffer::RebuildBuffer()
{
    int i,j;
    m_nNumOfTriangles = 0; // total number of vertex
    m_textureGroups.clear();
    DeleteDeviceObjects();

    /// add get the total number of triangles for each texture group
    int nCount = (int)m_pTerrain->m_CountStrips;
    for (i = 0; i < nCount; ++i)
    {
        int texID = m_pTerrain->m_pTriangleStrips[i].Setup(m_pTerrain);
        if(texID != INVALID_TEXTURE_ID)
        {
            set<TextureGroup, TextureGroup_less>::iterator iter;
            pair<set<TextureGroup, TextureGroup_less>::iterator, bool> res =
m_textureGroups.insert(TextureGroup(texID));
            (*(res.first)).nNumTriangles += m_pTerrain->m_pTriangleStrips[i].GetTriangleNum();
        }
    }
    nCount = (int)m_pTerrain->m_CountFans;
    for (i = 0; i < nCount; ++i)
    {
        int texID = m_pTerrain->m_pTriangleFans[i].Setup(m_pTerrain);
        if(texID != INVALID_TEXTURE_ID)
        {
            set<TextureGroup, TextureGroup_less>::iterator iter;
            pair<set<TextureGroup, TextureGroup_less>::iterator, bool> res =
m_textureGroups.insert(TextureGroup(texID));
            (*(res.first)).nNumTriangles += m_pTerrain->m_pTriangleFans[i].GetTriangleNum();
        }
    }
}

```

```

    }

    /// set the start triangle index of each texture group
    int nStartIndex=0;
    set<TextureGroup, TextureGroup_less>::iterator itCurCP, itEndCP = m_textureGroups.end();

    for( itCurCP = m_textureGroups.begin(); itCurCP != itEndCP; ++ itCurCP)
    {
        (*itCurCP).nStartIndex = nStartIndex;
        nStartIndex +=(*itCurCP).nNumTriangles;
        (*itCurCP).nNumTriangles = 0;
    }
    m_nNumberOfTriangles = nStartIndex;

    //
    // Create a vertex buffer that contains the vertex data sorted by texture cell group
    //
    if(m_nNumberOfTriangles>0)
    {
        IDirect3DDevice9* pD3dDevice = CGlobals::GetRenderDevice();

        pD3dDevice->CreateVertexBuffer(3*m_nNumberOfTriangles*sizeof(terrain_vertex),
            D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY, D3D9T_TERRAIN_VERTEX,
            D3DPPOOL_DEFAULT, &(m_pVertexBuffer), NULL);
        terrain_vertex *pVertices = NULL;

        m_pVertexBuffer->Lock( 0, 0, (void**)&pVertices, D3DLOCK_DISCARD );

        nCount = (int)m_pTerrain->m_CountStrips;
        for (j = 0; j < nCount; ++j)
        {
            TriangleStrip* pTri = m_pTerrain->GetTriStrip(j);
            if(pTri->IsEnabled())
            {
                set<TextureGroup, TextureGroup_less>::iterator iter =
m_textureGroups.find(TextureGroup(pTri->textureId));
                if(iter != m_textureGroups.end())
                {
                    int nTriCount = ((*iter).nStartIndex + (*iter).nNumTriangles)*3;
                    (*iter).nNumTriangles += pTri->BuildTriangles(m_pTerrain, pVertices, nTriCount);
                }
            }
        }
        nCount = (int)m_pTerrain->m_CountFans;
        for (j = 0; j < nCount; ++j)
        {
            TriangleFan* pTri = m_pTerrain->GetTriFan(j);
            set<TextureGroup, TextureGroup_less>::iterator iter =
m_textureGroups.find(TextureGroup(pTri->textureId));
            if(iter != m_textureGroups.end())
            {
                int nTriCount = ((*iter).nStartIndex + (*iter).nNumTriangles)*3;
                (*iter).nNumTriangles += pTri->BuildTriangles(m_pTerrain, pVertices, nTriCount);
            }
        }
        m_pVertexBuffer->Unlock();
    }
}

```

9.3 Summary and Outlook

Terrain is an important part for outdoor game engine. It may take up to 1/4 of the total rendering time of a moving frame in a game featuring advanced terrain rendering. Terrain rendering algorithm used in computer games usually uses several LOD to reduce the triangle count. However, an optimal LOD configuration is too CPU-time consuming to be used in games; hence game engine usually makes some compromise between optimal appearance and speed.

Nowadays, outdoor terrain engine is usually built on height map or elevation map, which is essentially a 2.5D terrain. By implementing terrain holes, we can divide game world to two parts (above terrain and below it). Other terrain topology may be researched in the future, such as a closed sphere or other more complex 3D shapes.

On the other side, outdoor terrain rendering in present game engines is far from realistic, due to the large dataset involved. Terrain rendering is still a very active research area.

Chapter 10 Ocean

This chapter implements large scale ocean simulation and rendering with reflection and refraction maps in ParaEngine. Water (including ocean) simulation and rendering is a well studied research area. The computation complexity of ocean rendering techniques differs greatly. More photo-realistic ocean rendering technique can now be introduced in computer game engine. In this chapter, we will see in detail one implementation which is suitable for current hardware. Although, there is some mathematics behind ocean rendering, the overall coding complex of a fair-looking ocean module is relatively low. But for games, we will need to deal with more conditions, such as rendering above and below the water surface, etc.

10.1 Foundation

In ParaEngine, the ocean module is called ocean manager. Typically, there are two sub modules in it.

- Simulation of the water surface (generating undulating height field on each frame)
- Rendering the water surface
 - a) forming triangle mesh for visualization of the surface
 - b) Texturing the mesh with optic effects (such as reflection, refraction)

In this section, we will examine the mathematics and programming techniques for each of the two sub modules.

10.1.1 Simulation of Water Surface

Water surface is simulated as a grid of height field and surface normal for each vertex.

```
struct sAnimVertex{  
    /** the height of the vertex */  
    float zPos;  
    /** the normal of the vertex */  
    Vector2 vNormal;  
};  
sAnimVertex m_Grid [ GRID_SIZE ] [ GRID_SIZE ];
```

In games, the shape of the ocean surface is mainly affected by two factors:

- disturbance of moving objects on surface, so that player swimming on the water surface will leave ripples behind
- winds, so that ocean surface never rest to peace

They are usually simulated or approximated using very different algorithms and combined in the final water surface presentation. Both can be done on GPU; but for the moment, we prefer the flexibility of doing them on CPU. In case of pure ocean simulation, only the second factor needs to be considered.

Simulation of disturbance of moving object on water surface

A simple way to simulate water is to solve the 2D Wave Equation on a uniform grid of points. It assumes that a moving object on surface will continuously generate 2D sinuous

waves that propagate on the surface. The 2D Wave Equation is $\frac{\partial^2 y}{\partial t^2} = c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2} \right)$, where x, z are 2D positions on the plane, y is the height (amplitude), the c term is the speed that the wave travels. What we would like to do is to integrate the left side of the equation using some numerical method so that, given the height field of y in previous frames, we can integrate over a small time step to get the new height field for the current frame. To be concise, we use the Verlet Integration Equation, which is given below.

$y(t_2) = 2 \cdot y(t_1) - y(t_0) + a(t_1) \cdot h^2$, where $y(t)$ is the height of a point at time t , $a(t)$ is the acceleration of the point at time t , and h is the time step, which should be a constant.

Verlet integration requires two initial positions to start integration and that the time step should be a constant. We will introduce a damping factor to absorb some excess energy that results from the inexact numerical integration. The equation now becomes:

$$y(t_2) = 1.99 \cdot y(t_1) - 0.99 \cdot y(t_0) + a(t_1) \cdot h^2 \quad (1)$$

The acceleration $a(t)$ at position (x, z) can be replaced by its numerical form as

$$\begin{aligned} a_{x,y} &= \frac{\partial^2 y}{\partial t^2} = c^2 \left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2} \right) \\ &= c^2 \left[(y_{x-1,z} + y_{x+1,z} - 2 \cdot y_{x,z}) + (y_{x,z-1} + y_{x,z+1} - 2 \cdot y_{x,z}) \right] \quad (2) \\ &= c^2 [y_{x-1,z} + y_{x+1,z} + y_{x,z-1} + y_{x,z+1} - 4 \cdot y_{x,z}] \end{aligned}$$

When implementing it, we will keep two height field maps in memory, one for the last height map $y(t_1)$, the other for the one before the last one $y(t_0)$. To compute the next height field $y(t_2)$, we first calculate the acceleration $a(t_1)$ with equation (2), and then calculate $y(t_2)$ with equation (1). This is rather straight forward. The normal for each point is calculated by its adjacent height values. At any time, we can disturb the water surface at any location by blending height field $y(t_1)$ with the shape of newly introduced disturbance sinusoidal wave at time step h .

The simulation described above can be effectively done in GPU, where the two height maps are kept as textures and processed in pixel shader. The source code can be found on NVIDIA's developer website.

Simulation of Ocean Waves

Unlike simple water surface simulation described above, wavy deep ocean surface are way too complex to be physically simulated. Hence, they are kind of "modeled" using statistical method. This is a much active research direction. One of the latest methods is based on using fast Fourier transformation (FFT) to produce tillable height map. This method is rather scalable and can be used not only for high-quality water animation for video, but also in real-

time applications. This is why we have chosen it as our ocean simulation method. There is also a good tutorial in the game community here¹⁴

The core idea of the FFT modeling method is that we assume that the surface waves are assembled from many linear waves generated by wind over an area much larger than the correlation length of the waves. The phases and amplitudes of these waves are in fact created by some predefined statistical function. For example, experimental measurements of surface-wave statistics confirmed that water surface descriptors have Gaussian distributions.

To implement this method, the height field of ocean surface is written as a summation of 2D sinusoidal waves of all kinds of frequency, i.e. as Fourier series.

$$h(x,z,t) = \sum_{u=-N/2}^{N/2-1} \sum_{v=-N/2}^{N/2-1} H(u,v,t) \cdot e^{\frac{i2\pi}{N}(u \cdot x + v \cdot z)} \quad (3)$$

$$H(u,v,t) = (H_0(u,v) \cdot e^{i\omega t} + H_0(-u,-v) \cdot e^{-i\omega t}) \quad (4)$$

$$\omega = \sqrt{g \cdot |K|} = \sqrt{g \cdot \sqrt{(2\pi u/N)^2 + (2\pi v/N)^2}} \quad (5)$$

We will explain these equations. The height field is presented by a $2^q \times 2^q$ grid, where $2^q = N$. $h(x,z,t)$ is the height of the point (x, z) at time t . $K = (2\pi u/N, 2\pi v/N)$ is a frequency domain vector, which can be understood as the direction of a 2D wave (u, v) traveling on the water plane; whereas $H(u, v, t)$ is a complex number, which can be interpreted as the magnitude and phase of the 2D wave (u, v) . The sigma integration means that the water plane height at position (x, z) is determined by summing up all these waves traveling on the ocean surface.

In (4), $H_0(u, v) = H(u, v, 0)$ is a table of initial amplitudes and phases at time $t=0$, which are generated by a random process. We use equation (5) to approximate the angular velocity for the wave (u, v) assuming ocean is very deep, where g is the gravitational constant.

$H(u, v, t)$ is the Fourier transform of $h(x, y, t)$. If, at any given time t , we can calculate $H(u, v, t)$ from equation (4) and (5) in the frequency domain, we will be able to transform it to $h(x, z, t)$ using Fast Fourier Transform (FFT).

The only unknown term left for calculating $H(u, v, t)$ is $H_0(u, v)$. $H_0(u, v)$ is generated by equation (6)

$$H_0(u, v) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P(\bar{k})} \quad (6)$$

$$P(\bar{k}) = A \frac{1}{k^4} e^{-l/(kL)^2} |\bar{k} \cdot \bar{w}|^2, \text{ where } \bar{k} = (u, v), k = |(u, v)|, L = |\bar{w}|^2 / g \quad (7)$$

$$(\xi_r, \xi_i) = \text{Guassian_random}(0, 1) \quad (8)$$

¹⁴ Lasse Staff Jensen and Robert Golias “Deep-Water Animation and Rendering”

http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm 2001

The coefficients of H_0 are Gaussian distributed random number with zero mean value and one standard deviation, shown in equation (8). The wave spectrum (magnitude for different frequency) is modeled by an analytical equation (7). The spectrum from this equation is called Phillip spectrum, which is initially used by Tessendorf for modeling wind-driven ocean waves. In equation (7), L is the largest possible wave arising from a continuous wind with speed w, g is the gravitational constant, w is direction of the wind and k is direction of the wave. A is a numeric constant globally affecting heights of the waves. The last term in the equation ($|\vec{k} \cdot \vec{w}|^2$) eliminates waves moving perpendicular to the wind direction.

For rendering the height field we need to calculate the gradient of the field to obtain normal. The traditional approach of computing finite difference between nearby grid points may be used, but it can be a poor approximation of the slope of waves with small wavelengths. Therefore, if we can afford it (in terms of computational power), it is better to use another FFT. This time evaluating the following sum:

$$n(x, y, t) = \nabla h(x, z, t) = \sum_{u=-N/2}^{N/2-1} \sum_{v=-N/2}^{N/2-1} N(u, v, t) \cdot e^{\frac{i2\pi}{N}(u \cdot x + v \cdot z)} \quad (9)$$

$$N(u, v, t) = i \cdot (u, v) \cdot H(u, v, t) \quad (10)$$

Now we have finished with the math model. To implement it in code, we need three complex tables (H , N , H_0) and one float table (W). The table H is for both $H(u, v, t)$ and $h(x, y, t)$. The imaginary components of H for $h(x, y, t)$ is always 0. The table N is for both $N(u, v, t)$ and $n(x, y, t)$. The table H_0 stores precalculated $H_0(u, v)$. The float table W stores pre-calculated angular velocity ω for all waves. The procedure of ocean simulation is given below.

```
Initialization(Ocean Parameters){
```

```
    The ocean is initialized with parameters such as wind vector and maximum wave height.
```

```
    // calculate  $H_0(u, v)$ 
```

```
    For each (u, v) {
```

```
        Generate two guassian random numbers:  $(\xi_r, \xi_i)$ : Equation (8)
```

```
        Calculate Phillip Spectrum at (u, v): Equation (7)
```

```
        Calculate  $H_0(u, v)$  and fill table item in  $H_0(u, v)$ : Equation (6)
```

```
        Calculate the angular frequency  $\omega$  at (u, v) and fill in table W: Equation (5)
```

```
    }
```

```
}
```

```
// called every frame
```

```
FrameMove(deltaTime){
```

```
    t = t+ deltaTime;
```

```
    (a) For each (u, v) {
```

```

Calculate H(u,v,t) and fill table item in H(u,v) : Equation (4)
Calculate N(u,v,t) and fill table item in N(u,v) : Equation (10)
}

// 2D Fast Fourier Transform

(b) H =Vertical_FFT(H);
(c) H =Horizontal_FFT(H);
(d) N =Vertical_FFT(N);
(e) N =Horizontal_FFT(N);
(f) // ... build mesh with H and N ...
// ... render the ocean ...
}

```

For performance reasons, we can distribute step (a) to (f) to separate frames, so the height table is only generated every 6 frames. For all intermediary frames when a height table is not available, we can linearly interpolate between the last height table and the one before the last one. Nowadays, step (a) to (f) can also be done by GPU.

10.1.2 Rendering of Water Surface

In previous sections, we have shown how to simulation height and normal for ocean vertices. In this section, we will look at how to build triangle meshes and rendering them with reflection and refraction maps.

Building Ocean Triangle Mesh

Some LOD algorithms like in terrain engine may be used. But usually it may not worth the additional computation on CPU. Instead, we prefer to use tiled brutal force rendering. The height field generated from ocean wave simulation is perfectly tiled, thus we can draw multiple instances of the same height field on tiled positions in the world. We will still need to remove ocean tiles that are not visible in the camera frustum. The good thing is that we only need to send one height table to GPU. This time, it is certainly good to use indexed triangle strips for the mesh, so that we can use fixed index buffers and only need to update the height and normal vertex buffers every few frames (remember that we can interpolate?).

Rendering with Optic Effects

Considering the reflections and refractions on the ocean surface from both above and below the water surface, one may misjudge the difficulty of rendering fair looking ocean. In fact, if our game engine has a configurable rendering pipeline, we can easily render ocean effects using less than a hundred lines of GPU shader code. But we will first look at the physics model behind. Here we only cover the physics of optic effects which we are going to simulate.

When light hits to a surface that separates two different materials (e.g. air, water), the ray is usually both reflected and refracted. Reflected portion of light returns to the first material mirrored by the surface normal n and the refracted part is transmitted into the second material. This is illustrated in Figure 10.1. Naturally, refraction is only modeled if the material is translucent. Physically, when light refracts, its direction is altered by the law of refraction, or Snell's law:

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$

n refers to the index of refraction and θ refers to the angle between the ray and surface normal. It should be noted that the change in the refracted ray's direction is often omitted or modeled with some approximate distortion function.

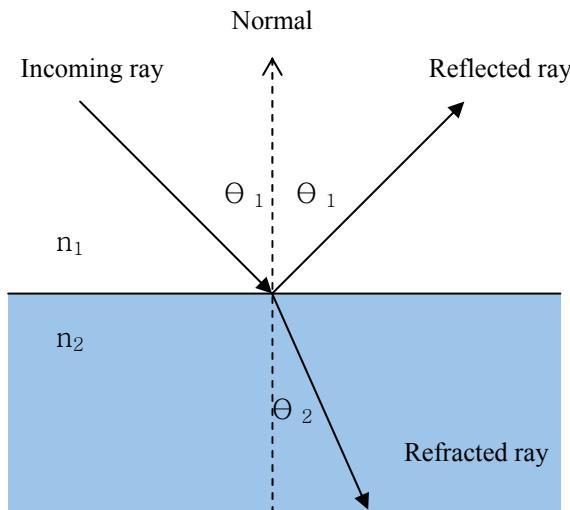


Figure 10.1 Reflection and refraction of light ray

Now, we will look at how reflection and refraction are modeled and then combined to form the final color.

Reflection Model:

The most common reflection model used today is the Phong reflection model, which is a combination of three components: ambient, diffuse and specular.

$$I_{reflect} = I_{ambient} + I_{diffuse} + I_{specular}$$

$$I_{diffuse} = k_d \sum_n I_n (L_n \cdot N)$$

$$I_{specular} = k_s \sum_n I_n (R_n \cdot V)^p$$

$$I_{ambient} = I_a \cdot k_a$$

k_d, k_s are the wavelength-dependent reflection and refraction coefficients for a material, I_n refers to the intensity of a point light and $(L_n \cdot N)$ is the cosine of the angle between the

direction of the light source and surface normal. ($R_n \cdot V$) is the cosine of the angle between viewing direction V and the mirror reflection direction R_n . p simulates surface roughness, effectively the “sharpness” or “glossiness” of the reflection. The summation is over all light sources. As for the ambient component, it is simply the product of the intensity of the ambient light I_a and material ambient property (k_a).

Adding them together, we have:

$$I_{reflect} = I_a \cdot k_a + \sum_n I_n (k_d (L_n \cdot N) + k_s (R_n \cdot V)^p)$$

$R_n = L_n - 2(N \cdot L_n) \cdot N$, which is a little inefficient to evaluate. Blinn introduces a new vector H called “half-vector”, which eliminates the calculation of R completely. It is defined as the unit normal to a surface that would reflect the light L_n along V and is mathematically much easier to calculate than R .

$$H = \frac{1}{2}(L + V)$$

By substituting R by H and adding a term for attenuation (f_{att}), based on the distance to the light source, we get the easy to evaluate reflection equation of Blinn-Phong model.

$$I_{reflect} = I_a \cdot k_a + \sum_n I_n f_{att} (k_d (L_n \cdot N) + k_s (H_n \cdot N)^p)$$

When computing the reflection component using GPU, the ambient and diffuse component is actually read from a pre-rendered reflection map, which is generated by mirroring the camera with the reflection plane and rendering the scene in to a reflection map texture. Finally the only specular light we will include is the light from the sun. Now the equation becomes:

$$I_{reflect} = reflection_map(TexCords(V)) + I_{sun} k (H_{sun} \cdot N)^p$$

Below is a procedure of rendering `reflection_map`: Reflect the original camera as shown in Figure 10.2. Render the whole scene (except water) from this camera to the so called “reflection” texture.

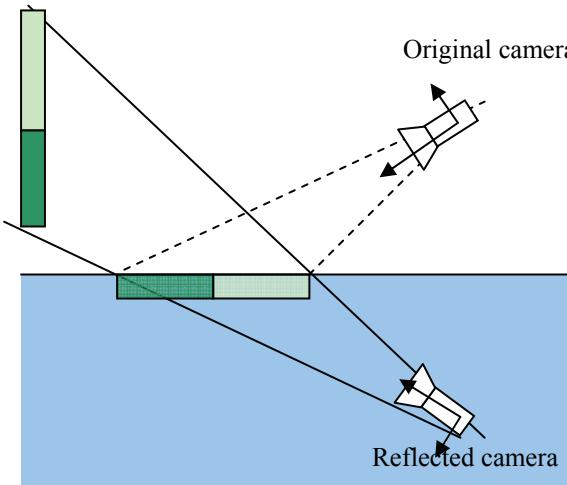


Figure 10.2 Camera reflection

Figure 10.3 shows the image taken from the original camera and the image take from the reflected camera. Please note that when rendering the reflection map, we will use a clip plane slightly below the water surface to prevent objects on the other side of water from being rendered in to the reflection map.

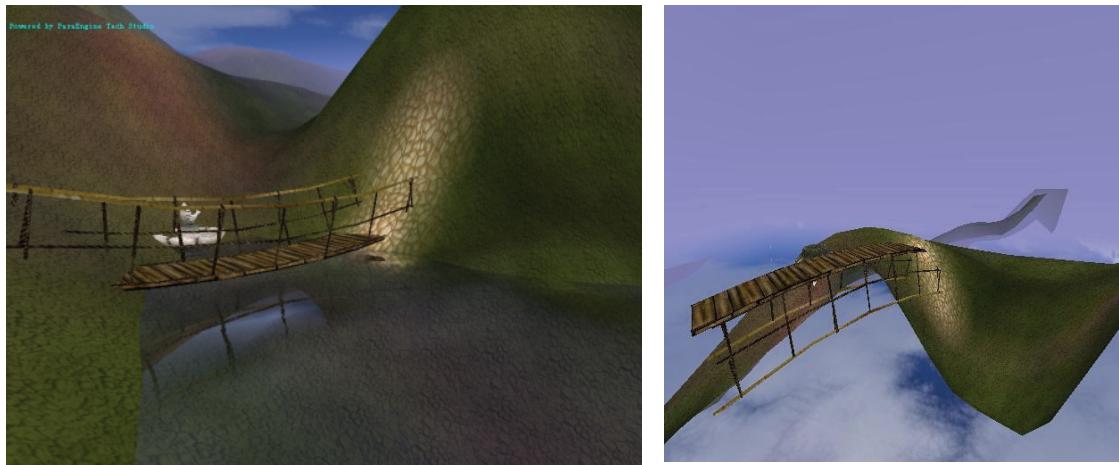


Figure 10.3 Original camera image (left) and reflection map (right)

This technique works fine for plane water. For oscillating water, we will use the same reflection texture but to shift the texture mapping position according to the normal of the surface point. This is of course an approximate and can be done in such way: in every grid point of the wave, we calculate the reflection ray of its view vector in world coordinate system, shift a small fixed distance along this ray to obtain a new position, and then project it to screen space to obtain the shifted texture coordinates. So instead of rotating the water plane, we can imagine that the water plane is shifted by a fixed distance to form another parallel reflection plane. This is just an approximation. See Figure 10.4. In Figure 10.3, we will notice that the right image is distorted using the above method when mapped to the water surface in the left image. We will show the actual code later.

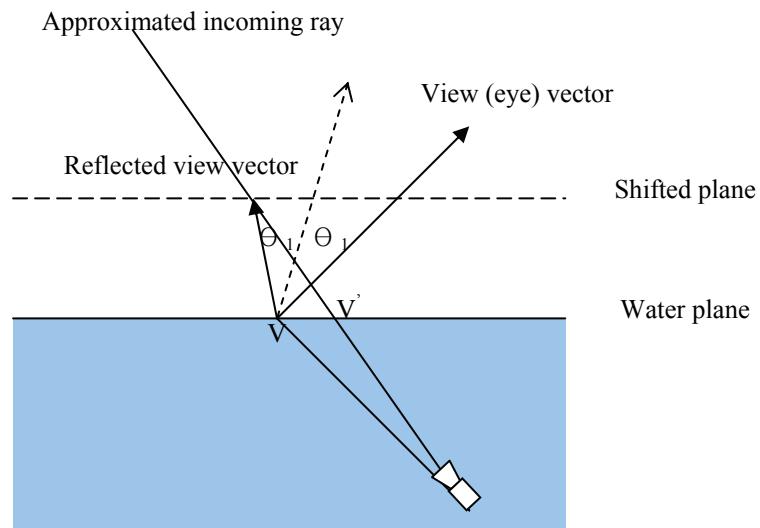


Figure 10.4 Normal correction

The last problem is that reflection and refraction on wavy water surface usually show more scenes than on still water surface. This will result to texture misses at the image edge as well as the place where water meets the land. To cover texture misses, we increase FOV (field of view) for the cameras used for the reflection map. In most cases, 10% larger than the main camera's FOV is enough. In case there are still texture misses even with increased field of view, we use clamping texture addressing mode. Clamping addressing mode ensures that the missing texture have the same color of the edge of the texture.

Refraction Model:

The refraction map is generated similar to the reflection map using the unreflected original camera. The refraction map may also be distorted by the surface normal and indices of the two materials. However, if we do not need the optic effects of refraction map distortion, we can ignore the use of a refraction map and use alpha blending instead. We will see this later.

Mixing the Refraction and Reflection:

Once we have obtained the refraction and reflection color of a point on the water surface, we will need to mix them according to the amount of energy passed and reflected by the surface. This ratio is determined by Fresnel coefficient. The original physics formula is complex enough.

$$F(k) = \frac{(g - k)^2}{2(g + k)^2} \left(1 + \frac{(k(g + k) - 1)^2}{(k(g - k) + 1)^2} \right), \text{ where } k = -\vec{V} \cdot \vec{N}, g = \sqrt{\left(\frac{n_2}{n_1}\right)^2 + k^2 - 1}, \text{ and } V$$

is the view vector; N is the normal of the surface.

We approximate it with the function:

$G(k) = (1 - \vec{V} \cdot \vec{N})^{-\alpha}$, where α is coefficient corresponding to relative refraction index (see the following table). When rendering for games, we usually use a small α .

n2/n1	A
1.1	10
1.2	8
1.33	7
1.5	6

The final formula we will use is this one:

$$\text{FinalColor} = \text{Refraction} + F(k) * (\text{Reflection} - \text{Refraction}) + \text{Specular}$$

In case a total reflection happens, we can eliminate the refraction component. The actual code is given in the next section.

10.2 Architecture and Code

10.2.1 Ocean Rendering Pipeline

The render pipeline is given below.

```
For each frame(){  
    ...  
    If(scene moves){  
        Reflect the camera by the water plane  
        Set clip plane and Render scene into the reflection texture  
        Set the camera back to the original one  
        [optional] Set clip plane and Render scene into the refraction texture  
    }  
    Render the scene to back buffer.  
    ...  
    RenderOcean(){  
        Animate height and normal field  
        Render the ocean mesh with reflection and refraction map  
    }  
}
```

There are two places to be careful during coding:

- (1) When the camera is reflected, objects and triangles which are initially invisible may become visible in the new camera. For example, the terrain engine will need to re-tessellate and rebuild the triangle mesh according to the reflected camera view frustum.
- (2) When rendering the reflection map, one can choose to reflect either the scene objects or the camera. For example, one can build a reflection matrix P and pre-multiply it with the original view matrix V ; this will reflect all scene objects when the new view matrix is applied. The disadvantage of this approach is that it makes the new view matrix meaningless, e.g. all triangle normals are reversed, which means that all meshes must reverse their default triangle winding direction when rendering the reflection map. Alternatively, we can reflect the camera eye, lookat and up vector and rebuild a valid camera view matrix. In this way, triangle winding is not affected and that we always have a meaningful view matrix. However, the reflection image should be flipped horizontally in the image space. Fortunately, this can be done easily in vertex shader or pixel shader. So we advise to mirror the camera instead of all scene objects when rendering the reflection map.

10.2.2 Shader code for Reflection and Refraction mapping

The following shader code shows the reflection and refraction mapping described in this paper. The vertex shader interpolates between two sets of height and normal values to obtain the grid point's vertex position and normal in world coordinate system. The computeTexCoord() function distorts the texture coordinate for the reflection map and it also adjust it according to the FOV of the reflection map.

Please note that we did not use a refraction map in this shader code. Instead we use alpha blending with the render target (i.e. the backbuffer). In order to use the render target as the refraction map, we must render the scene (except for water) first, before the water is rendered. In the pixel shader, we will need to generate the reflection and specular color, as well as an alpha value by which the new pixel color is blended with color on the render target. If one uses a refraction map, the code will be more straight-forward: just turn off alpha blending, and add the reflection, refraction and specular term together to form the final pixel color.

```
// Desc: Based on the Vertex Texture Fetch paper on Nvidia developer's website.  
//@def whether use backbuffer as refraction map.If so,  
// pixel color alpha value is used as aprox. fresnel term.  
#define ALPHA_BLENDING_REFRACTION_MAP  
  
// parameters  
const float4x4 mViewProj: VIEWPROJECTION: register(vs, c10);  
const float4 vWorldPos: worldpos;  
const float4 vCameraPos: worldcamerapos;  
const float4 sun_vec: sunvector;  
//const float4 sun_color: suncolor;  
const float4 posOffset : posScaleOffset;  
const float4 vsReflectionTexcoordScaleOffset:uvScaleOffset;  
const float4 psFresnelR0: FresnelR0;  
const half3 psTotalInternalReflectionSlopeBias : ConstVector0;  
const float4 vsNormalSign : ConstVector1;  
  
texture tex0 : TEXTURE;  
// reflection map  
sampler reflectionSampler = sampler_state  
{  
    texture = <tex0>;  
    AddressU = clamp;  
    AddressV = clamp;  
    MINFILTER = LINEAR;  
    MAGFILTER = LINEAR;  
};  
  
struct Interpolants  
{  
    float4 positionSS : POSITION;      // Screen space position  
    float2 reflectionTexCoord : TEXCOORD0; // texture coordinates for reflection map  
    float3 normalWS : TEXCOORD1;       // normal in world space  
    float3 eyeVectorWS : TEXCOORD2;    // eye vector in world space  
    float3 halfVectorWS : TEXCOORD3;   // half vector in world space  
};  
/////////////////////////////  
//  
//          Vertex Shader  
//
```

```

///////////////////////////////
// Vertex shader Function declarations
float2 computeTextureCoord(float4 positionWS, float3 directionWS, float distance, float2 scale, float2 offset);

Interpolants vertexShader( float2 Pos      : POSITION,
                           float ZPos0   : POSITION1,
                           float2 Norm0   : NORMAL0,
                           float ZPos1   : POSITION2,
                           float2 Norm1   : NORMAL1)
{
    Interpolants o = (Interpolants)0;
    float4 positionWS;
    float4 flatPositionWS;
    float3 normalWS;
    float3 reflectionVectorWS;
    float2 nXY;
    float1 waveHeight;
    // offset xy and interpolate z to get world position
    flatPositionWS = float4(Pos.x+posOffset.z, vWorldPos.y, Pos.y+posOffset.w, 1);
    waveHeight = lerp(ZPos0, ZPos1, posOffset.x);
    positionWS = flatPositionWS;
    positionWS.y += waveHeight*0.2; // make smaller wave

    // transform and output
    o.positionSS = mul(positionWS, mViewProj);

    // Output the world space normal
    nXY = lerp(Norm0, Norm1, posOffset.x);
    normalWS = normalize(float3(nXY, 8.0f).xzy); // LXZ: why 8.0f, 24.0f are all OK values
    normalWS.xyz *= vsNormalSign.x; // Flip the normal if we're under water
    o.normalWS = normalWS;

    // Output the eye vector in world space
    o.eyeVectorWS = normalize(vCameraPos - positionWS);

    // Output the half vector in world space
    // No need to normalize because it's normalized in the pixel shader
    o.halfVectorWS = o.eyeVectorWS + sun_vec;

    // Calculate reflection map coordinates

    // Compute the reflection vector in world space
    reflectionVectorWS = reflect(-o.eyeVectorWS, normalWS);

    // Compute the reflection map coordinates
    o.reflectionTexCoord.xy = computeTextureCoord(positionWS, reflectionVectorWS, 0.4f,
vsReflectionTexCoordScaleOffset.xy, vsReflectionTexCoordScaleOffset.zw);

    return o;
}
// computeTextureCoord() takes a starting position, direction and distance in world space.
// It computes a new position by moving the distance along the direction vector. This new
// world space position is projected into screen space. The screen space coordinates are
// massaged to work as texture coordinates.
float2 computeTextureCoord(float4 positionWS, float3 directionWS, float distance, float2 scale, float2 offset)
{

```

```

float4 positionSS;

// Compute the position after traveling a fixed distance
positionWS.xyz = positionWS.xyz + directionWS * distance;
positionWS.w = 1.0;

// Compute the screen space position of the newly computed position
positionSS = mul(positionWS, mViewProj);

// Do the perspective divide
positionSS.xy /= positionSS.w;

// Convert screen space position from [-1,-1]->[1,1] to [0,0]->[1,1]
// This is done to match the coordinate space of the reflection/refraction map
positionSS.xy = positionSS.xy * 0.5 + 0.5;

// Account for the fact that we use a different field of view for the reflection/refraction maps.
// This overdraw allows us to see stuff in the reflection/refraction maps that is not visible
// from the normal viewpoint.
positionSS.xy = positionSS.xy * scale + offset;

// Flip the t texture coordinate upside down to be consistent with D3D
positionSS.y = 1 - positionSS.y;

// Return the screen space position as the result. This will be used as the texture coordinate
// for the screenspace reflection/refraction maps.
return(positionSS.xy);
}

///////////////////////////////
//
//          Pixel Shader
//
///////////////////////////////

half computeFresnel(half3 light, half3 normal, half R0);

half4 pixelShader(Interpolants i) : COLOR
{
    half4 specular;
    half4 reflection;
    half4 refraction;
    half fresnel;
    half4 o;

    // Normalize direction vectors
    i.normalWS = normalize(i.normalWS);
    i.halfVectorWS = normalize(i.halfVectorWS);

    // Compute the specular term
    specular.x = pow(max(dot(i.halfVectorWS, i.normalWS), 0), 3);

    // Put a cliff in the specular function
    if(specular.x < 0.5)
    {
        specular.x = 2.0 * specular.x * specular.x;
    }
    specular.xyz = specular.xxx * half3(0.2, 0.2, 0.2);
    specular.w = 0;
}

```

```

// Do the texture lookup for the reflection

//reflection = tex2D(reflectionSampler, i.reflectionTexCoord.xy);
reflection = tex2D(reflectionSampler, float2(1-i.reflectionTexCoord.x, i.reflectionTexCoord.y));

// Do the texture lookup for the refraction
...
// Handle total internal reflection
half refractionVisibility = saturate(dot(i.eyeVectorWS, i.normalWS));
refractionVisibility = saturate((refractionVisibility - psTotalInternalReflectionSlopeBias.y) *
psTotalInternalReflectionSlopeBias.x +
                                psTotalInternalReflectionSlopeBias.z);
// Give some blue tint to the refraction
// refraction = lerp(refraction, half4(0, 0, 1, 0), .1);

// Compute the fresnel to blend the refraction and reflection terms together
fresnel = computeFresnel(i.eyeVectorWS, i.normalWS, psFresnelR0.x);

#ifndef ALPHA_BLENDING_REFRACTION_MAP
// Combine the refraction,a blue tint(0,0,0.1), reflection and specular terms
half alpha = 1-refractionVisibility*0.9*(1-fresnel);
o.xyz = reflection.xyz*fresnel+(specular.xyz + (1-alpha)*half3(0, 0, 0.1))/alpha;
o.w = alpha;
return o;
#else
// Combine the refraction, reflection and specular terms
return(lerp(refraction, reflection, fresnel) + specular);
#endif
}

half computeFresnel(half3 eye, half3 normal, half R0)
{
    // R0 = pow(1.0 - refractionIndexRatio, 2.0) / pow(1.0 + refractionIndexRatio, 2.0);

    half eyeDotNormal;
    eyeDotNormal = dot(eye, normal);

    // Make sure eyeDotNormal is positive
    eyeDotNormal = max(eyeDotNormal, -eyeDotNormal);

    return(R0 + (1.0 - R0) * pow(1.0 - eyeDotNormal, 4.5));
}

///////////////////////////////
//
//          Technique
//
/////////////////////////////
technique OceanWater_vs20_ps20
{
    pass P0
    {
        ZENABLE = TRUE;
        ZWRITEENABLE = TRUE;
        ZFUNC = LESSEQUAL;
    }
}

```

```

AlphaBlendEnable = true;
AlphaTestEnable = false;

// shaders
VertexShader = compile vs_2_0 vertexShader();
PixelShader = compile ps_2_0 pixelShader();
}

}

```

Figure 10.5 is an image generated with the above code without using a refraction map.



Above water surface

Below water surface

Figure 10.5 Above and under water scene without refraction map

10.3 Summary and Outlook

Ocean simulation and rendering techniques used in games can already generate approaching photorealistic scenes. What we covered in this chapter is only one set of popular approaches. Still, there are a few other add-on things we have missed, such as the shorelines, bump mapping and under water effects, etc. Shorelines can be created by blending a special shoreline texture to the water surface, according to the depth of the water (see the following section). Bump mapping can give you the effects of high-frequency wavelets on the water surface. Wavy underwater effects can be achieved by a pixel shader in the image space after the scene has been rendered. Water interaction with the physical environment (fluid dynamics) is also an area which we believe future game engines will be featuring.

10.3.1 Shorelines

We will end this chapter by the implementation of shorelines. Please see Figure 10.6. Shorelines can be easily implemented by comparing the water level and the terrain height at each rendered pixel of the ocean surface. We will use the difference between the water and terrain level at each pixel to obtain a blending factor, which blends an animated shoreline texture with that of the original ocean color. In the figure, a static white bitmap is used as the shoreline texture to illustrate how it is blended with the original ocean surface.

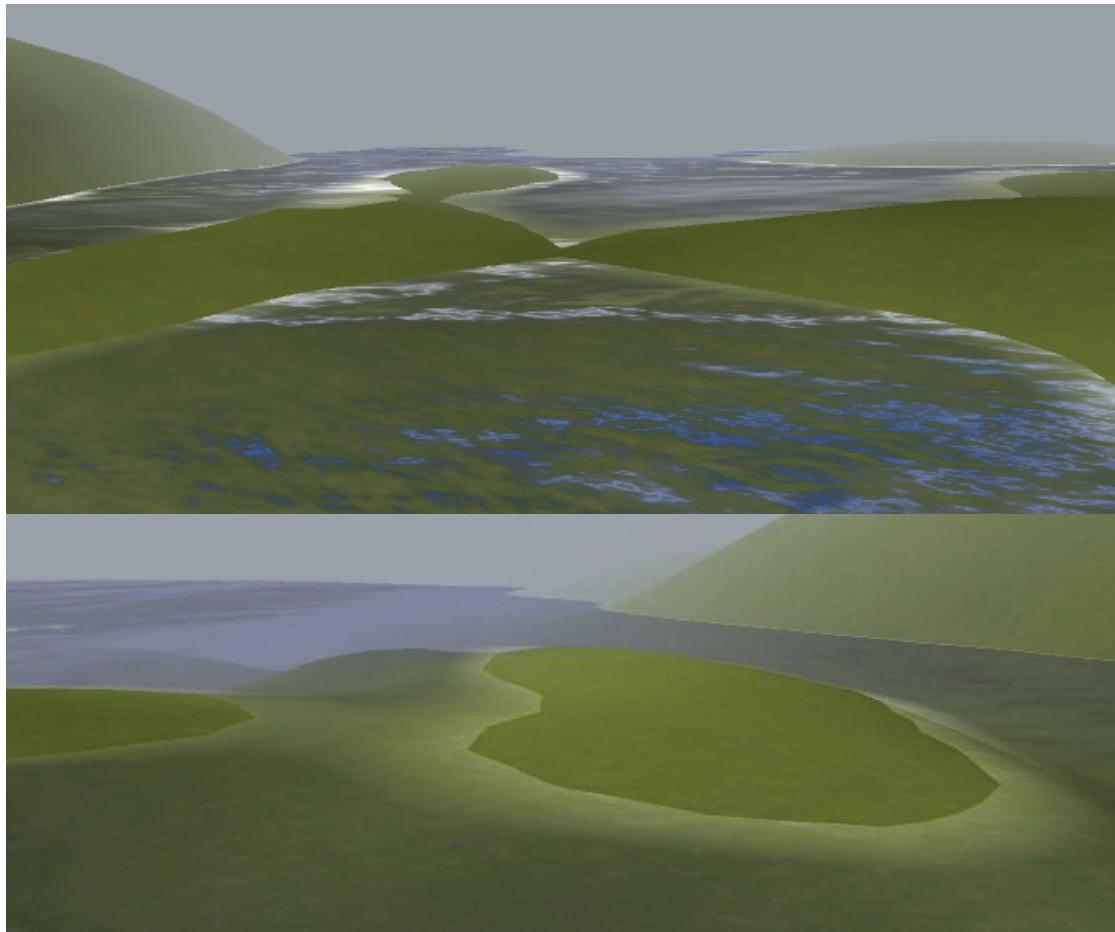


Figure 10.6 Shore lines

The above procedure can be carried out in shader programs. In the vertex shader input, we add an additional terrain height value for each ocean surface vertex. The vertex shader will output the height difference between terrain height and the water height for each vertex. This height difference is automatically interpolated when passed to the input of a pixel shader. In the pixel shader, we will use it to obtain an alpha blending factor, using the code below.

```
blendfactor = (i.heightDifference+0.5)/0.5;
if(i.heightDifference >0)
    blendfactor = 1;
if(i.heightDifference <-0.5)
    blendfactor = 0;
o.xyz = lerp(o.xyz, half3(1,1,1), alpha); // half3(1,1,1) may be replaced by a shoreline texture
```

`i.HeightDifference` is the height difference passed to the input of a pixel shader. `Blendfactor` is the linear interpolation factor to blend the shoreline texture and the original ocean color. 0.5 in the above code is arbitrarily chosen as the maximum shoreline water depth, beyond which the shoreline texture will not be painted to the ocean surface.

Chapter 11 Special Effects

In this chapter, we talk about a few special effects which are commonly seen in present-day game engines. They are shadow, light map, environment map, local lighting, image post processing, per-pixel fog, billboarding and reflective surfaces. It should be pointed out that there are generally no standard ways to integrate them in to a game engine. Each of them has one or several basic techniques behind. If you are implementing a new game engine from scratch, we advise that you go over the basic ideas of these techniques, but postpone their implementations until you have a stable and robust game engine framework or a funded game project.

The implementation of special effects may be easily out-dated due to the fast advancement of graphics hardware and techniques. It is better to consider them as add-on things to the game engine and plug-in them to the rendering pipeline whenever and wherever necessary.

This chapter is organized by effects.

11.1 Shadows

Shadow is one of the most expensive rendering effects in games. There are also a large number of different techniques to generate them; yet there are no perfect ones.

11.1.1 Static Shadow

The least expensive way is to bake the shadow to model and terrain textures or model vertex colors, when the game scene is made by the artists and the level designers. This special texture layer generated for shadow is commonly referred to as light map, whose colors are usually multiplied with the original model color to darken the shadowed region. At runtime, the game engine computes nothing; it just renders the models as usual. The biggest advantage of this approach is that it is fast at runtime. The biggest problem is that it does not work efficiently with dynamic objects or dynamic light sources, such as an animated character or a moving light. Due to the high cost of other shadow techniques, static pre-generated shadow is still commonly used for shadows cast from static scene objects and with a relatively static light source. For example, the shadows of terrain and buildings under sun light.

11.1.2 Dynamic Shadow

The focus of this section is on other shadow techniques suitable for dynamic shadows. Many of the shadow generation techniques developed over the years have been used successfully in offline movie production. It is still challenging, however, to compute high quality shadows in real-time for dynamic scenes. We will only briefly discuss some methods suitable for interactive applications. Table 11.1 shows a comparison of some commonly used real-time shadow rendering algorithms. Please refer to Woo et al.'s paper¹⁵ for a broader survey of shadow algorithms.

¹⁵ Andrew Woo, Pierre Poulin and Alain Fournier, A survey of shadow algorithms, IEEE Computer Graphics and Applications, vol 10 (November 1990), pp. 13-32.

Table 11.1 Shadow rendering algorithm comparison

Plane Projected Shadows	Projected Shadows	Depth Shadow Mapping
Quick, not much calculations High detail No self-shadowing no shadow receivers	Quick, almost no calculations Detail depends on texture No self-shadowing Shadow receivers	Very quick, not much calculations Detail depends on texture Self-shadowing Shadow receivers
Vertex Projection	Shadow Volumes	Smoothies
Slow with high-res meshes High Detail No self-shadowing no shadow receivers	Slow, lots of calculations High detail Self-shadowing Shadow receivers	Very quick, not much calculations Detail depends on Smoothies Self-shadowing Shadow receivers

In DirectX SDK, one can find source code samples using two different shadow algorithms. One is called shadow mapping; the other is called shadow volume. Shadow mapping is easy to implement and fast, but has so-called aliasing artifacts; shadow volume is computationally more expensive and difficult to implement, but can generate very robust (i.e. fewer artifacts) and good looking (even soft-edged) shadows. We will briefly explain shadow mapping and go in to some depth of shadow volume.

11.1.2.1 Shadow mapping

Shadow mapping is perhaps the most ubiquitous and easy-to-implement shadow algorithms. Its concept is straightforward. See Figure 11.1. The entire scene is drawn twice (in two render passes). In the first pass, the scene is rendered to a texture from the light's perspective. This texture is called a shadow map and is generated from simple vertex and pixel shaders. For each pixel, the pixel shader writes the pixel depth instead of the pixel color. The same pixel may be overwritten several times, so that when the entire scene has been rendered, the shadow map will contain the distances from the light position to the scene geometries. In the second pass, the scene is rendered from the camera's perspective: the distance between each pixel and the light is generated once again and compared to the corresponding depth generated previously in the shadow map. If they match, then the pixel is not in shadow. If the distance in the shadow map is smaller, the pixel is in shadow, and the shader can darken the pixel color accordingly. The most suitable type of light to use with shadow maps is a spotlight since a shadow map is rendered by capturing the scene from a point camera position.

The major advantage of using shadow maps over shadow volumes is efficiency since a shadow map only requires the scene to be rendered twice. There is no geometry-processing needed and no extra mesh to generate and render. An application using shadow maps can therefore maintain good performance regardless of the complexity of the scene.

However, the shadow quality using shadow mapping is limited by the resolution of the shadow map texture and affected by the relative positions of the camera and light. E.g. blocky shadow edges or distinctly jagged edges frequently appear between shadowed and unshadowed regions due to under-sampling of the shadow map. There are improved versions to

the original shadow volume algorithm, such as the perspective shadow mapping, which can be found in NVIDIA developer network.

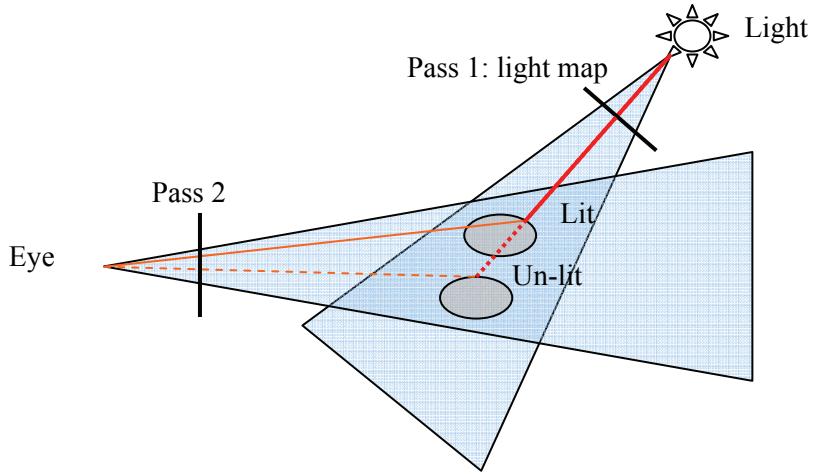


Figure 11.1 Shadow mapping

There are several things to take care about when implementing shadow mapping in the game engine.

First thing is to build a complete list of shadow receivers and shadow casters in the rendering pipeline. We must not forget to include shadow caster objects outside the camera view frustum, because their shadows may be casted in the view frustum. A quick solution is to use objects within a certain radius of the current camera eye position to build the list of shadow casters. More over, we can use some predefined value such as 50, to limit the total number of shadow casters. The code in the render pipeline is given below.

```
if( m_nMaxNumShadowCasters <(int)sceneState.listShadowCasters.size()
{
    // remove shadows casters that are far from the eye position.
    sceneState.listShadowCasters.sort(LessPostRenderObj_NoTechBatch<PostRenderObject>());
    int nRemoveNum = (int)sceneState.listShadowCasters.size() - m_nMaxNumShadowCasters;
    list<PostRenderObject>::iterator itFrom = sceneState.listShadowCasters.end();
    for( int i=0;i<nRemoveNum;++i)
        --itFrom;
    sceneState.listShadowCasters.erase(itFrom, sceneState.listShadowCasters.end());
}
// only for effect file batch
sceneState.listShadowCasters.sort(LessPostRenderObj<PostRenderObject>());
```

The selection of shadow receivers also plays an important role in shadow mapping. A fixed-sized shadow map must contain the bounding box of all shadow receivers, so the shadow map quality is directly relevant to the size of the bounding box of shadow receivers. The choice of shadow receivers is dependent on the camera mode: if the game is mainly a top-down view, we can include both the terrain and objects on the terrain as shadow receivers; if it is a free camera mode, it is sometimes only good to cast shadows on a small region of terrain near the camera. On either case, terrain is the most important shadow receiver.

When rendering into the shadow map, there is another thing to take care of. It is texture alpha testing. We need to emulate alpha testing using alpha blending in the pixel shader if the shadow map render target does not support alpha testing.

Figure 11.2 shows a scene rendered with shadow map (the tree leaves used texture alpha testing); Figure 11.3 shows the shadow map generated for it. One can find the shader code in the code section.



Figure 11.2 A 3D scene rendered with shadow mapping in ParaEngine

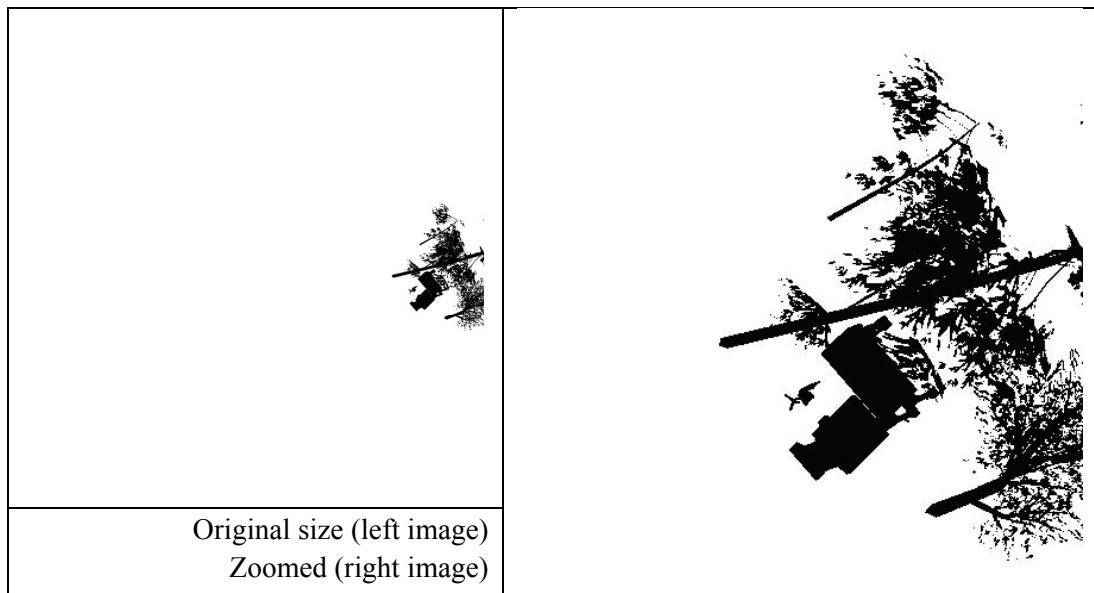


Figure 11.3 1024*1024 sized shadow map generated from the light's perspective

11.1.2.2 Shadow volume

In shadow volume technique, the scene objects may be either shadow caster or shadow receiver or both. The computation complexity of shadow volume is mostly dependent on the number of shadow casters and has nothing to do with the number of shadow receivers. So if the game scene has a large number of shadow receivers (such as the terrain and buildings) and just a few shadow casters (the main characters), shadow volume can generate perfect shadows efficiently. In an actual game, one may combine the use of static shadow with that of shadow volumes, such that shadows cast by terrain and buildings are pre-made in light maps, whereas dynamic shadows of characters are generated using shadow volumes. Moreover, shadow volume is also suitable for multiple light sources. The quality of shadows using shadow volume is very high and independent of light position and shadow receivers. See Figure 11.4

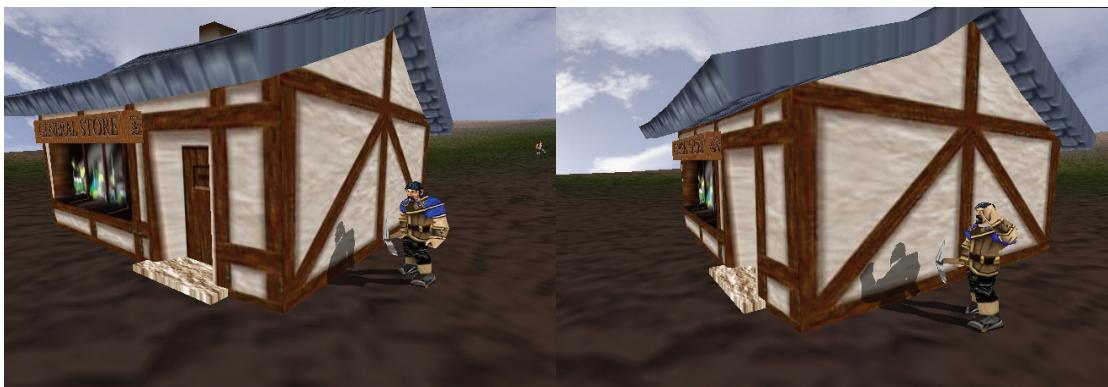


Figure 11.4 Shadow volume rendering results

Before we go on, there are some limitations to the shadow volume method.

There are quite a few limitations to shadow volumes, in many respects they are the medium-level of shadow rendering. They aren't the fastest (planar shadows generally are) and they don't look the best (soft shadowing/projective shadows generally look better). However, for the speed and features we get, they are probably the most practical to implement currently.

An overview of the main limitations:

1. Hardware Stencil Buffering

The application requires the use of a stencil buffer. We need as many bits for the stencil buffer as possible; one may be able to work with a 1-bit stencil, but ideally either a 4 or an 8 bit stencil buffer are preferred for more accurate shadow rendering. This is because I will render all shadow volumes in a batch to the stencil buffer. If these volume geometry overlap after projection to the 2D space, it will cause the stencil buffer to overflow.

2. Bandwidth Intensive

The algorithm will chew up as much graphics card bandwidth as you can throw at it, especially when it comes to rendering with multiple light sources. Fill rate (number of pixels rendered per second) in particular is very heavily used; with a possible $n+1$ overdraw (where n is the number of lights). There are a few tricks one can use to reduce the trouble this causes.

3. No Soft Shadowing

The mathematical nature of a shadow volume dictates that there are no intermediary values. It is a Boolean operation. Pixels are either in shadow or not. Therefore one can often see distinct aliased lines around shadows.

4. Complicated for Multiple Light Sources

The majority of real-time scenes have several lights (4-5) enabled at any one point in time, whilst with this technique there is no limit to the number of lights (even if the device caps indicate a fixed number). The more lights that are enabled the slower the system goes. The two factors to watch out are geometric complexity (and number of meshes) and light count, the shadow rendering system usually uses an algorithm to select only the most important lights, and only the affected geometry for rendering.

Implementation

Shadow volume algorithm imposes an additional constraint for models that are animated by skinning (the process of blending different transformations of a vertex from nearby bones). The process used in hardware to skin the model must be identically replicated in software, so the possible silhouette determination step can find the possible silhouette of the animated model, not the static pose. This is a problem common to all shadow volume techniques and cannot be eliminated until graphics hardware is sophisticated enough to support the adjacency data structures needed for possible silhouette determination. In ParaEngine, skinned animation is by default calculated in software; if shadow volume is enabled, it becomes the only option for skinned animation.

Shadow Rendering Pipeline

1. Render the portion of the scene which will receive shadows.
2. For each active light in the scene. {
 - 2.1. Build light-camera pyramid for Z-Fail testing
 - 2.2. For each shadow caster {
 - 2.2.1. Decide whether to use Z-fail or Z-pass algorithm by checking the bounding box of the model with the light-camera pyramid.
 - 2.2.2. Build shadow volume for the current shadow caster, with regard to the current light.
If Z-fail is used, shadow volume must be capped at both front and back ends
 - 2.2.3. Render the shadow volumes to stencil buffer twice to mark the shaded region. Render method is either Z-fail or Z-pass.
 - }
 - 2.3. Render the shadow (i.e. alpha blending a big quad of the shadow color to the screen), with regard to the stencil buffer
- }
3. Render the portion of the scene that does not receive shadows.

The following order of game scene rendering is adopted in the game engine.

1. all shadow receivers: terrain, mesh object, etc
2. **shadow volumes** of shadow casters
3. shadow casters
4. anything that does not receive shadows, such as particles, GUI, etc.

Since shadows will only be cast on objects rendered previously in the pipeline, characters will not cast shadows on themselves, if we draw them after their shadows. Due to the nature

of the shadow volume algorithm, the transparent region in textures will appear to be fully shaded on the 3D model.

Shadow Volumes Calculation

In ParaEngine, any shadow caster class need to implement the ShadowCaster interface. For example, the biped scene object is a shadow caster. In its shadow caster implementation, it calls the BuildShadowVolume function of its associated model class. This class has the geometry data of the 3D model, which will be used to build the Shadow Volume. Shadow volume is built by firstly finding the silhouette edges (please refer to related papers on this), and then extruding from the model's pivot point to infinity along the opposite light direction. See Figure 11.5.

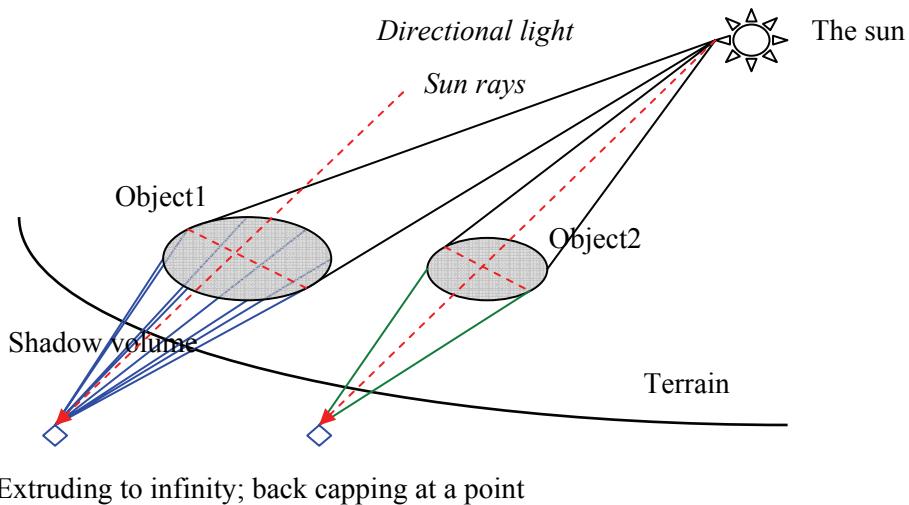


Figure 11.5 Shadow Volume Calculation for Directional Light

The above shadow is incomplete, because its front cap is open. If the object is not transparent and the camera is not in the shadow volume, this uncapped shadow volume will suffice for rendering accurate shadows with Z-pass algorithm. However, when the *near* plane of the camera's frustum intersects the shadow volume, we should swap to another computationally more extensive algorithm (z-fail). Complementary to Z-pass, Z-fail algorithm only works when the shadow volume is closed (capped) at both ends and the *far* plane of the camera does not intersect the back cap. Since the light's angle to the ground is limited to well above zero in the game engine. We can eliminate the need to check if the back cap is outside the far plane of the camera's frustum. However, we must close the shadow volume's front cap which is why z-fail algorithm is computationally expensive. The algorithm in Table 11.2 is used decide when to swap to z-fail algorithm. The rules are labeled from (a) to (e).

Table 11.2 Z-Fail Testing algorithm

If(model does not have bounding box or sphere){
(a) we will not test screen distance. i.e. we will draw its shadow anyway

```

}else{
    (b) Check if object is too small after projection to 2D screen. If so, its shadow will not be
        rendered};

If(model must be rendered using Z-Fail){ (c) use Z-fail}

Else if(model intersects with Occlusion pyramid){ (d) use Z-fail}

Else{ (e) use z-pass};

```

Occlusion pyramid is defined to be the shape formed by the four vertices of the camera's near plane and the sun position. Figure 11.6 shows some sample test cases. Bounding box 1 and 2 will use Z-fail algorithm by the rule (d). Bounding box 4 will not cast shadows by the rule (b). Bounding box 3 will use z-pass by the rule (e).

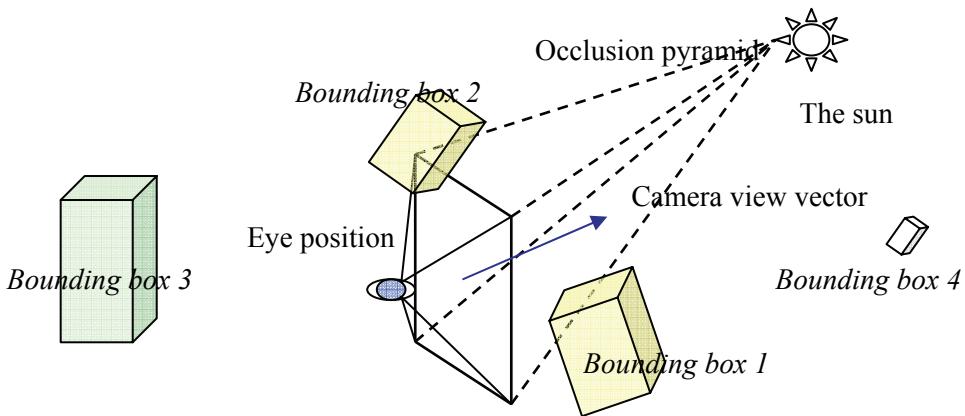


Figure 11.6 Z-Fail algorithm testing

Performance analysis

The real bottlenecks in a stencil shadow volume implementation are silhouette determination and shadow volume rendering. The former requires a huge amount of CPU cycles and it worsens if the occluders had high polygon counts. The latter is a big consumer of invisible fill rate. One way to alleviate the CPU burden during silhouette determination is to use a lower polygon model of the occluder. Another effective way is to determine a new silhouette only every 2-4 frames. This is based on the assumption that the light's position or the occluder's position does not change very drastically within 2-4 frames. This assumption turns out to be pretty good for most cases. Another good trick is to pre-calculate all static shadow volumes. Moreover, shadow extrusion may also be done in HLSL shader.

11.1.3 Code

Some shadow mapping shader code is given below.

For each shadow casters, its associated effect file must contain a shadow technique which renders the object into the shadow map. The code is usually identical and is given below.

```

void VertShadow( float4 Pos      : POSITION,
                 float2 Tex      : TEXCOORD0,
                 out float4 oPos  : POSITION,
                 out float2 outTex : TEXCOORD0,
                 out float2 Depth : TEXCOORD1 )
{
    oPos = mul( Pos, mWorldViewProj );
    outTex = Tex;
    Depth.xy = oPos.zw;
}

float4 PixShadow( float2 inTex     : TEXCOORD0,
                  float2 Depth     : TEXCOORD1 ) : COLOR
{
    half alpha = tex2D(tex0Sampler, inTex.xy).w;

    if(g_bAlphaTesting)
    {
        // alpha testing
        alpha = lerp(1.0, alpha < ALPHA_TESTING_REF);
        clip(alpha-0.5);
    }
    float d = Depth.x / Depth.y;
    return float4(d.xxx, alpha);
}

technique GenShadowMap
{
    pass p0
    {
        VertexShader = compile vs_2_a VertShadow();
        PixelShader = compile ps_2_a PixShadow();
        FogEnable = false;
    }
}

```

For each shadow receiver, we need to generate shadow texture coordinates in vertex shader, and sample the shadow map in the pixel shader. The code is usually identical and is given below. Please note, we have given the code for both hardware shadow map and traditional Float 32 render target.

```

texture ShadowMap2 : TEXTURE;
sampler ShadowMapSampler: register(s2) = sampler_state
{
    texture = <ShadowMap2>;
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = None;
    AddressU = BORDER;
    AddressV = BORDER;
    BorderColor = 0xffffffff;
};

Interpolants vertexShader( float4 Pos      : POSITION,
                           float3 Norm   : NORMAL,
                           float2 Tex     : TEXCOORD0)
{
    Interpolants o = (Interpolants)0;

```

```

// ... other code ignored
    if(g_Useshadowmap)
    {
        o.tex1 = mul(Pos, mLightWorldViewProj);
    }
    return o;
}

half4 pixelShader(Interpolants i) : COLOR
{
    half4 o = {0,0,0,1};
    // ... other code ignored
    if(g_Useshadowmap)
    {
        if(g_UseShadowmapHW)
        {
            // hardware shadow map
            half3 shadow  = tex2Dproj(ShadowMapSampler, i.tex2).rgb;
            //colorDif = ((shadow*0.2+0.8) * colorDif)*0.9 + 0.1;
            colorDif = (shadow*0.25+0.75) * colorDif;
        }
        else
        {
            // F32 shadow map
            float2 shadowTexCoord = i.tex2.xy / i.tex2.w;
            float shadowTestDepth = i.tex2.z / i.tex2.w;

            #ifdef MULTI_SAMPLE_SHADOWMAP
                // transform to texel space
                float2 texelpos = g_nShadowMapSize * shadowTexCoord;

                // Determine the lerp amounts
                float2 lerps = frac( texelpos );

                //read in bilerp stamp, doing the shadow checks
                //shadowTestDepth = shadowTestDepth- SHADOW_EPSILON;
                color0.x = (tex2D( ShadowMapSampler, shadowTexCoord ) >= shadowTestDepth);
                texelpos = shadowTexCoord + float2(1.0/g_nShadowMapSize, 0);
                color0.y = (tex2D( ShadowMapSampler, texelpos ) >= shadowTestDepth);
                texelpos = shadowTexCoord + float2(0, 1.0/g_nShadowMapSize);
                color0.z = (tex2D( ShadowMapSampler, texelpos ) >= shadowTestDepth);
                texelpos = shadowTexCoord + float2(1.0/g_nShadowMapSize, 1.0/g_nShadowMapSize);
                color0.w = (tex2D( ShadowMapSampler, texelpos ) >= shadowTestDepth);

                // lerp between the shadow values to calculate our light amount
                float shadow = lerp( lerp( color0.x, color0.y, lerps.x ),
                    lerp( color0.z, color0.w, lerps.x ),
                    lerps.y );
                lerps.x=(shadowTexCoord.x<0 || shadowTexCoord.y<0 || shadowTexCoord.x>1 ||
shadowTexCoord.y>1);
                shadow = lerp(shadow, 1, lerps.x);
            #else
                float shadowDepth = tex2D(ShadowMapSampler, shadowTexCoord);
                float shadow      = (shadowTestDepth <= shadowDepth);
            #endif
            colorDif = (shadow*0.25+0.75) * colorDif;
        }
    }
}

```


11.2 Light Mapping

Ever since the days of Quake, developers have extensively used light mapping to achieve realistic lighting almost without any computing overhead at runtime. However, light mapping technique is slowly becoming out dated as computer hardware evolves and real-time per pixel lighting becomes possible.

Implementing light map requires two steps (1) light map generation and (2) applying light maps to geometry. The first step is a complicated and time-consuming (so far, not real-time) process. It does not make much sense to integrate it to the game engine unless you are working on a powerful world editor for level designers. Instead, we can rely on other professional editing tools, such as 3dsmax, Maya, etc to generate them for a game scene. In this section, we will only focus on how light map related information can be retrieved from these tools, and how to apply them to geometry.

11.2.1 Light Mapping Basics

The idea of light mapping is very simple. In addition to the main diffuse texture, it maps another texture (so called light map) to all faces of the mesh. Each triangle face of the mesh is mapped to a portion of the lightmap. At realtime, a pixel on the model face is determined by the multiplication of the diffuse texture color and light map color. Figure 11.7 shows a light mapped quad mesh.

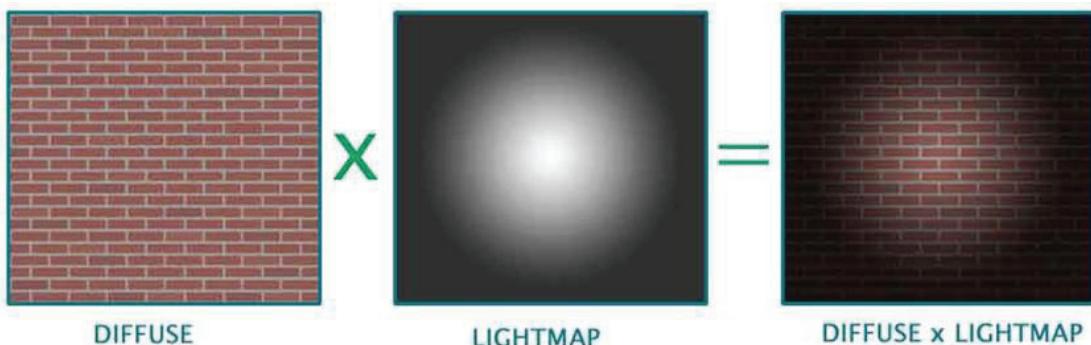


Figure 11.7 Light map basics¹⁶

Figure 11.8 shows a VR scene using light maps in ParaEngine. The buildings in the image are rendered with light maps, except for the windows which use a cubic environment map (see [section 11.3](#)) of the sky. The front statue also uses a light map, which emulates the self-shadow on it. The trees are billboarded (see [section 11.4](#)). You can compare the light mapped building with the one rendered without light map in Figure 11.9.

¹⁶ Taken from http://www.flipcode.com/articles/article_lightmapping.shtml by Keshav Channa

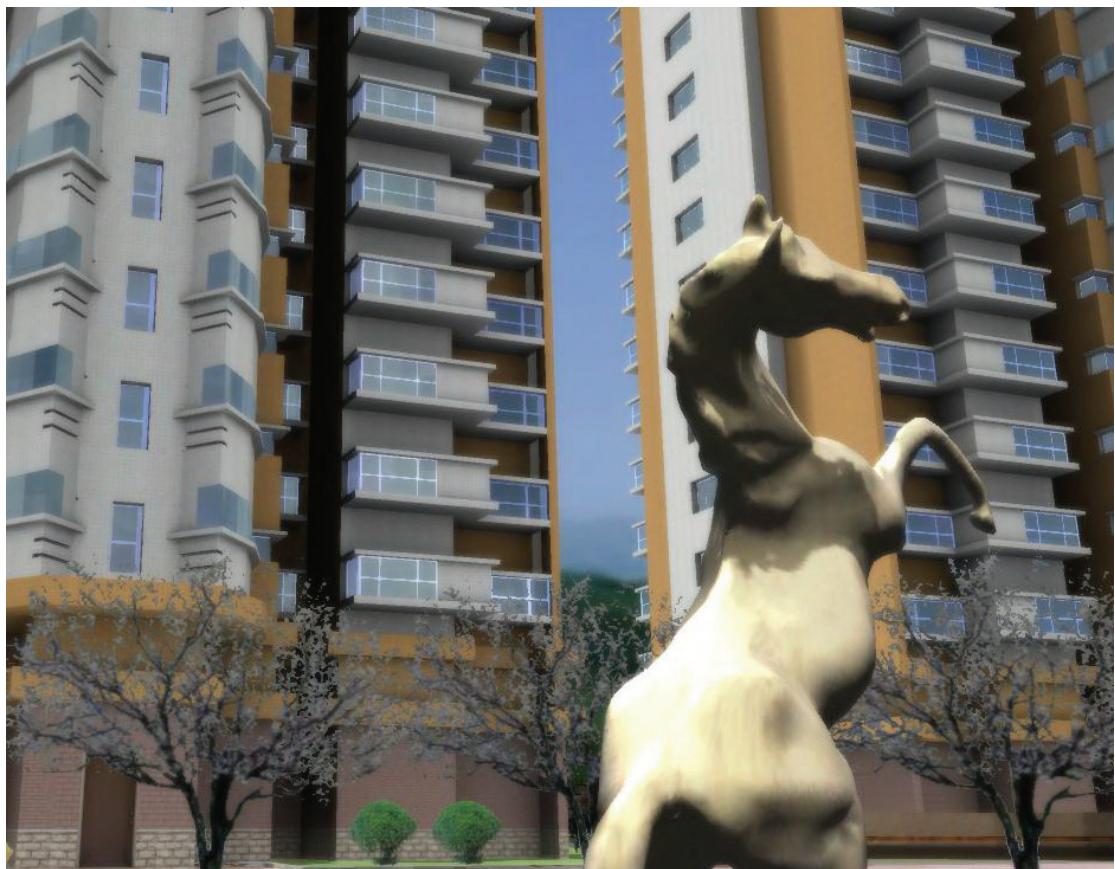


Figure 11.8 Light map example in ParaEngine

11.2.2 Consider Baked Texture First

Before using light maps, one should always consider baking lights to the main diffuse texture first (i.e. using the `diffuse*lightmap` as the diffuse map as shown in Figure 11.7). If this works in your situation, why bother to use an additional light map? If the mesh model is small or its main diffuse texture is not reused across mesh faces, using baked texture is usually a better choice. Light effect can be written to the main diffuse texture either by artists or by ray tracing tools provided in 3dsmax or Maya, etc.

11.2.3 When to Use Light Map

Baked texture has its many limitations. One limitation that leads us to use light map is that it consumes too much texture memory for large mesh models. Large mesh models, such as walls and streets usually reuse the same set of textures across many mesh faces. It is very wasteful to map unique high resolution textures to each mesh faces. Instead, we will reuse the diffuse texture for faces in the mesh model (and possibly in other models as well), but use a unique low-resolution texture (light map) which solely encode the brightness of each mesh face.

The quality of light effect is thus very dependent on the resolution of the light map. Fortunately, stretching light map does not induce many glitches and is sometimes preferred to create smooth shadow edges and brightness transitions.

11.2.4 Cost of Using Light Maps

Generally speaking, using light map will double or triple the total texture file size in the scene, and it will increase the vertex buffer size, since the light map uses another texture coordinate set and causes many mesh vertices to split. When there is only a single diffuse texture coordinate set, a mesh vertex only need to be split when the faces sharing the vertex, map it to different locations on the diffuse texture. Yet, with a second texture coordinate set for the light map, a vertex shared by multiple faces will usually need to be splitted, since the model always maps each of its face to a different location on the light map.

11.2.5 Retrieving Light Map Texture Coordinates from 3dsmax

In 3dsmax, light map texture coordinates are in another map channel of the mesh. In ParaEngine, we assume that it is in the second map channel. The process of retrieving it is given below.

```
For each face in the mesh{  
    For each of the three face vertex in the face{  
        Export position, normal, color, etc.  
        For each map channel of the mesh{  
            get the index of the vertex in the map channel  
            Export the texture coordinates of the vertex using its index.  
        }  
        Export skinning information if any.  
    }  
}
```

Mesh data exported from 3dsmax is then saved to model files. Besides a proprietary model file format, most game engine also supports a public extensible file format. If it is a DirectX based engine, it is usually the DirectX's X-file format. In ParaEngine, we support almost everything except for animated models in the original X-file format. The original X-file templates provide a field called FVFData, which can be used to store any FVF per-vertex data in the file. In case of light maps, the second UV coordinate set is saved in FVFData field. The DirectX extenstion library also support multiple UV sets through its X-file format (by the FVFData field). The only problem is that the original X-file templates do not provide a field to store the light map file name, so we have to do it manually in some other places or derive it from the main texture file name in the material template.

The following shows an example X file.

<pre> xof 0303txt 0032 // Exported by ParaEngine X file exporter V0.1 Mesh mesh_object{ 24; // 24 vertices -1.583985;0.000000;-1.284059;; 2.540000;0.000000;1.115350;; -1.583985;0.000000;1.115350;; 2.540000;0.000000;-1.284059;; -0.715449;2.324428;-0.897879;; 1.902904;2.324428;0.696656;; 1.938948;2.324428;-0.890688;; -0.690824;2.324428;0.719860;; -1.583985;0.000000;-1.284059;; 1.938948;2.324428;-0.890688;; 2.540000;0.000000;1.284059;; -0.715449;2.324428;-0.897879;; 2.540000;0.000000;-1.284059;; 1.902904;2.324428;0.696656;; 2.540000;0.000000;1.115350;; 1.938948;2.324428;-0.890688;; 2.540000;0.000000;1.115350;; -0.690824;2.324428;0.719860;; -1.583985;0.000000;1.115350;; 1.902904;2.324428;0.696656;; -1.583985;0.000000;1.115350;; -0.715449;2.324428;-0.897879;; -1.583985;0.000000;-1.284059;; -0.690824;2.324428;0.719860;; 12; // 12 faces 3;0,1,2;; 3;1,0,3;; 3;4,5,6;; 3;5,4,7;; 3;8,9,10;; 3;9,8,11;; 3;12,13,14;; 3;13,12,15;; 3;16,17,18;; 3;17,16,19;; 3;20,21,22;; 3;21,20,23;; MeshNormals { 24; // 24 normals 0.000000;-1.000000;0.000000;; 0.000000;-1.000000;0.000000;; 0.000000;-1.000000;0.000000;; 0.000000;-1.000000;0.000000;; 0.000000;1.000000;0.000000;; 0.000000;1.000000;0.000000;; 0.000000;1.000000;0.000000;; 0.000000;1.000000;0.000000;; 0.001378;0.164829;-0.986321;; 0.000000;0.166861;-0.985980;; 0.000865;0.165585;-0.986195;; 0.001378;0.164829;-0.986321;; 0.966159;0.257815;0.008211;; 0.964430;0.264338;0.000000;; </pre>	<p>A light mapped box like mesh</p> <p>24 Vertex Positions</p> <p>There are actually only 8 unique vertex positions. However, due to light mapping, each vertex is split three times for the three faces sharing it.</p> <p>12 triangle faces</p> <p>Each face indexes into the vertex array.</p> <p>24 normals for each vertex</p>
--	---

<pre> 0.966744;0.255505;0.011109;; 0.966159;0.257815;0.008211;; 0.004557;0.173885;0.984755;; 0.000000;0.167735;0.985832;; 0.002875;0.171617;0.985160;; 0.004557;0.173885;0.984755;; -0.935226;0.354012;0.005265;; -0.936743;0.350019;0.000000;; -0.934617;0.355581;0.007338;; -0.935226;0.354012;0.005265;; 12; // 12 faces 3;0,1,2;; 3;1,0,3;; 3;4,5,6;; 3;5,4,7;; 3;8,9,10;; 3;9,8,11;; 3;12,13,14;; 3;13,12,15;; 3;16,17,18;; 3;17,16,19;; 3;20,21,22;; 3;21,20,23;; } MeshMaterialList { 1; // Number of unique materials 12; // Number of faces 0,0,0,0,0,0,0,0,0,0,0,0;; Material { 0.000000;0.000000;0.000000;1.000000;; 64000.0; 0.900000;0.900000;0.900000;; 1.000000;0.000000;0.000000;; TextureFilename {"model/test/lightmap/0-ZHUA~1.dds";} } }// materials MeshTextureCoords { 24; // 24 UV 1.000000;0.971927;; 0.034312;0.031192;; 0.993762;0.024954;; 0.031192;0.962569;; 0.031192;0.962569;; 0.993762;0.024954;; 1.000000;0.971927;; 0.034312;0.031192;; 0.232200;0.782655;; 0.702227;0.308705;; 0.761080;0.774808;; 0.257704;0.277317;; 0.392147;0.677684;; 0.587364;0.448217;; 0.609382;0.658112;; 0.379915;0.424975;; 0.232200;0.782655;; 0.702227;0.308705;; 0.761080;0.774808;; 0.257704;0.277317;; </pre>	<p>Mesh materials</p> <p>Currently, the original DirectX template only supports a single texture file name per-vertex. However, we can either encode the light map file name into this texture file name or derive from it.</p> <p>24 UV coordinates</p> <p>This is the first set of UV coordinate for the diffuse texture.</p>
---	---

<pre> 0.392147;0.677684;; 0.587364;0.448217;; 0.609382;0.658112;; 0.379915;0.424975;; }// UV FVFData { 256; // FVF code 48; 1056325316,1052602860, 995417344,1035768352, 1056325316,1035768352, 995417344,1052602860, 1056551874,1053374800, 1008981770,1059832668, 1008981770,1053374800, 1056551874,1059832668, 1065185444,1060668198, 1057170974,1065185444, 1057170974,1060668198, 1065185444,1065185444, 1061833938,1038936608, 1057170974,1053103490, 1057170974,1038936608, 1061833938,1053103490, 1056551874,1060599939, 1008981770,1065117184, 1008981770,1060599939, 1056551874,1065117184, 1061929694,1053741612, 1057266730,1059870356, 1057266730,1053741612, 1061929694,1059870356; }// FVF }// mesh </pre>	<p>FVF per vertex data</p> <p>256 is the FVF code for TEX1 which means that the following data will be an array of (float, float) for each vertex.</p> <p>We use the FVFData field to store the 24 UV coordinates for the light map. Data is saved in DWORD type during text-encoding. Although it is difficult to read by human, it can be parsed very fast by the game engine.</p> <p>One can also move other per-vertex data to the FVFData field to slightly shorten the loading time of the mesh file. This hold true for both text and binary encoding.</p>
---	---

11.3 Cubic Environment Mapping

Cubic environment mapping technique is widely used for efficiently rendering reflective or semi-reflective surfaces in a static scene, such as the metallic reflections on the body of a teapot or a car. In essence, environment mapping is a method of addressing texture maps using a normalized vector. Hence, it can also be used to create many interesting user-defined visual effects.

Unlike spherical mapping, cubic environment mapping can be generated on the fly during game play. With hardware support, cubic environment mapping is now the preferred environment mapping method for games.

There are already extensive articles talking about environment mapping, including the DirectX SDK samples. We will not cover it in details. Instead we will only point out some limitations and common mistakes during implementation.

Attention on plane surfaces

If the normals of the mesh faces do not vary much, such as on a plane surface, the surface will map to a very small region on the environment map, resulting in terrible stretch of the

environment mapped textures. Special coding may alleviate the problem, but generally speaking, we should avoid applying environment map on large plane surfaces.

Attention on cubic map resolution

There is no need to use high resolution texture for cubic environment map, because it is mapped on a curved surface, making it hard to distinguish. But we need to enable mipmapping for the environment maps.

Obtaining the eye-reflection vector

Eye reflection vector is used to address the cubic map. Hence the eye reflection vector should be generated in the same coordinate system as the one used during cubic environment map generation. In most cases, we use the world coordinate system to generate the cubic environment map and so does the eye reflection vector. The pixel and vertex shader code is given below.

In Vertex shader

```
if(g_bEnvironmentMap)
{
    // Obtain the eye vector in world (cube) space
    float3 eyeVector = normalize( worldPos-g_EyePositionW );
    // Compute the reflection vector and save to tex1
    o.tex1 = normalize(reflect(eyeVector, worldNormal));
}
```

In Pixel shader

```
if(g_bEnvironmentMap)
{
    half4 reflection = texCUBE( texCubeSampler, i.tex1 );
    normalColor.xyz = lerp(normalColor.rgb, reflection.rgb, g_bReflectFactor);
}
```

11.4 Billboard

Billboarding is used for rendering objects which are always facing the camera. Sometimes, we can rotate all three axes to orient a mesh to the camera, as we did for particles in the particle system. Sometimes, we only rotate the Y axis (or the world up axis) to orient a mesh to the camera, such as billboarded trees, grasses, or even 3D objects, see Figure 11.9.



Figure 11.9 Billboaded trees

The downside of billboarding is that it does not look real at close distance and that dynamic shadows and physics does not apply to them. Billboarding is rarely used for big objects in games now, but is still widely used in other virtual reality applications.

11.5 Reflective Surfaces

Reflective or semi-reflective surfaces, such as floors, mirrors, or still water can be achieved using the same method as we did with the ocean surface. The only difference is that we need to share a fixed number of reflection maps for all reflective plane objects in the scene. For example, if there are many reflective mesh faces at different height levels, we can choose to render reflection maps for the closest visible surfaces.

Figure 11.10 shows an indoor room with semi-reflective floors rendered in ParaEngine, for a VR application. The building in the figure has many floors, however, the camera is only closet to the current floor; and so only the reflective surface on the current floor needs to be rendered.



Figure 11.10 Reflective surface

11.6 Local Lights

Local lights are lights which only affect a local region of the scene. A local light may be a point light or a spot light. All local lights have a radius property. A game world may contain any number of local lights. However, for performance reasons, there can only be very limited number of active lights for a given scene geometry. For each scene geometry, such as a character or a mesh, the game engine needs to automatically select the best lights which are affecting (near) it. The number of best lights can be limited to some predefined value, such as 4 or 8. The light system works with both fixed function pipeline and programmable pipeline. See Figure 11.11.

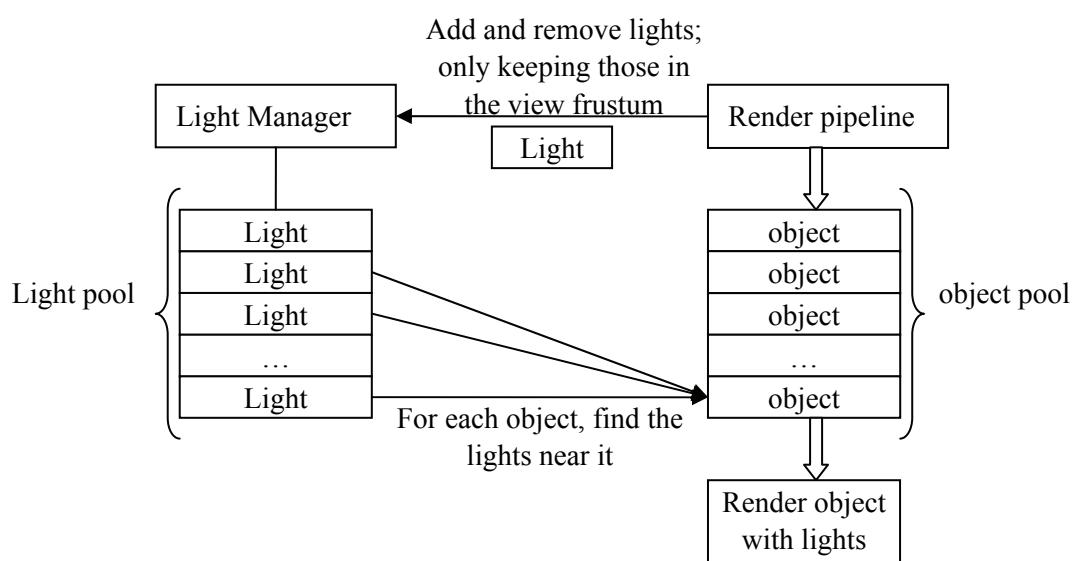


Figure 11.11 Local Light and render pipeline



Figure 11.12 Local lights example in ParaEngine

Figure 11.12 shows an example of local lights in ParaEngine. We will explain the light effects in the figure. There is one mesh object, two characters and two local lights in the scene. The mesh object is a bar with tables and props. The shadows on the wall and the floor are baked in texture. Two dog characters are standing at the side of two tables, on top of which two local lights are positioned. One is yellow, the other is green. We have marked the light positions using two crystal helper meshes. The bar mesh is lit by the two local lights, as shown by the color of the tables in it. The two mobile characters are lit by their closest local light respectively. So one is under the range of the yellow light, and the other is under the influence of the green light.

11.7 Full Screen Glow Effect

Full screen glow effect is an image space technique to give the entire scene a fancy look. Glow effect is both very simple and very universal, the latter means that it can be applied to any stage of the rendering pipeline. There is a paper explaining it very well. It is Real-Time Glow by Greg James and John O'Rorke on Gamasutra, May 26, 2004. Please take a good at the Figure 11.13 for the visual difference before and after full screen glow effect is applied.

Full screen glow is a simple version of those various visual effects created by high dynamic range lighting (HDR). The good news is that it can work with hardware multi-sampling (i.e. hardware anti-aliasing) and it does not even require a floating point render target to achieve its effect. The basic idea of full screen glow is given directed in the following steps. We will see later that all steps can be fit into one simple HLSL shader technique with multiple render passes.

Because it is an image space technique, we assume one already has the scene rendered in the back buffer or in whatever other render-targets.

- Copy the data from the current render target (in most cases, the back buffer) to a temporary texture render target of a smaller size (usually half the original size).
- Blur this texture image using a 5*5 or 9*9 filter. Here we use a Gaussian filter, and blur horizontally first and then vertically.
- Finally, blend this blurred image with the one on the original render target, and we are done.





Top is the final image with glow effect; bottom one is original image without glow effect.

Figure 11.13 Full screen glow effect

Figure 11.14 shows the intermediary images generated between the two in Figure 11.13.





Figure 11.14 Intermediary images in full screen glow effect

11.7.1 Code

Here is the shader code to perform those steps. Please note that the first pass in the technique is not necessary if one does not encode the glow factor in to the alpha channel of the render target, as discussed in the paper given at the start of this section.

```
// TexellIncrement should be set to 1/RRT_width and 1/RTT_height respectively for the two seperate convolution blur passes.
const float2 TexellIncrements : ConstVector0;
// Glowness controls the intensity of the glow. 0 means no glow, 1 is normal glow, 3 is three times glow.
const float4 Glowness : ConstVector1;

// texture 0
texture GlowMap0 : TEXTURE;
sampler GlowSamp1 : register(s0) = sampler_state
{
    texture = <GlowMap0>;
    AddressU = CLAMP;
    AddressV = CLAMP;
    AddressW = CLAMP;
    MIPFILTER = NONE;
    MINFILTER = LINEAR;
    MAGFILTER = LINEAR;
};

// texture 1
texture GlowMap1 : TEXTURE;
sampler GlowSamp2 : register(s1) = sampler_state
{
    texture = <GlowMap1>;
    AddressU = CLAMP;
    AddressV = CLAMP;
    AddressW = CLAMP;
    MIPFILTER = NONE;
    MINFILTER = LINEAR;
    MAGFILTER = LINEAR;
};
```

```

struct VS_OUTPUT_BLUR
{
    float4 Position : POSITION;
    float4 Diffuse : COLOR0;
    float4 TexCoord0 : TEXCOORD0;
    float4 TexCoord1 : TEXCOORD1;
    float4 TexCoord2 : TEXCOORD2;
    float4 TexCoord3 : TEXCOORD3;
    float4 TexCoord4 : TEXCOORD4;
    float4 TexCoord5 : TEXCOORD5;
    float4 TexCoord6 : TEXCOORD6;
    float4 TexCoord7 : TEXCOORD7;
    float4 TexCoord8 : COLOR1;
};

struct VS_OUTPUT
{
    float4 Position : POSITION;
    float4 Diffuse : COLOR0;
    float4 TexCoord0 : TEXCOORD0;
};

VS_OUTPUT VS_GlowSource(float3 Position : POSITION,
                       float3 TexCoord : TEXCOORD0)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
    OUT.Position = float4(Position, 1);
    OUT.TexCoord0 = float4(TexCoord, 1);
    return OUT;
}

VS_OUTPUT VS_Quad(float3 Position : POSITION,
                  float3 TexCoord : TEXCOORD0)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;
    OUT.Position = float4(Position, 1);
    OUT.TexCoord0 = float4(TexCoord, 1);
    return OUT;
}

VS_OUTPUT_BLUR VS_Quad_Vertical_5tap(float3 Position : POSITION,
                                      float3 TexCoord : TEXCOORD0)
{
    VS_OUTPUT_BLUR OUT = (VS_OUTPUT_BLUR)0;
    OUT.Position = float4(Position, 1);

    float3 Coord = float3(TexCoord.x, TexCoord.y, 1);
    float TexellIncrement = TexellIncrements.y;
    OUT.TexCoord0 = float4(Coord.x, Coord.y + TexellIncrement, TexCoord.z, 1);
    OUT.TexCoord1 = float4(Coord.x, Coord.y + TexellIncrement * 2, TexCoord.z, 1);
    OUT.TexCoord2 = float4(Coord.x, Coord.y, TexCoord.z, 1);
    OUT.TexCoord3 = float4(Coord.x, Coord.y - TexellIncrement, TexCoord.z, 1);
    OUT.TexCoord4 = float4(Coord.x, Coord.y - TexellIncrement * 2, TexCoord.z, 1);
    return OUT;
}

VS_OUTPUT_BLUR VS_Quad_Horizontal_5tap(float3 Position : POSITION,
                                       float3 TexCoord : TEXCOORD0)
{

```

```

VS_OUTPUT_BLUR OUT = (VS_OUTPUT_BLUR)0;
OUT.Position = float4(Position, 1);

float3 Coord = float3(TexCoord.x, TexCoord.y, 1);
float TexelIncrement = TexelIncrements.x;
OUT.TexCoord0 = float4(Coord.x + TexelIncrement, Coord.y, TexCoord.z, 1);
OUT.TexCoord1 = float4(Coord.x + TexelIncrement * 2, Coord.y, TexCoord.z, 1);
OUT.TexCoord2 = float4(Coord.x, Coord.y, TexCoord.z, 1);
OUT.TexCoord3 = float4(Coord.x - TexelIncrement, Coord.y, TexCoord.z, 1);
OUT.TexCoord4 = float4(Coord.x - TexelIncrement * 2, Coord.y, TexCoord.z, 1);
return OUT;
}

///////////////////////////////
//
//          Pixel Shader
//
/////////////////////////////

float4 PS_GlowSource(VS_OUTPUT IN) : COLOR
{
    float4 tex = tex2D(GlowSamp1, float2(IN.TexCoord0.x, IN.TexCoord0.y));
    tex.xyz = tex.xyz * tex.w;
    return tex;
}

// For two-pass blur, we have chosen to do the horizontal blur FIRST. The
// vertical pass includes a post-blur scale factor.

// Relative filter weights indexed by distance from "home" texel
// This set for 5-texel sampling
#define WT5_0 1.0
#define WT5_1 0.8
#define WT5_2 0.4

#define WT5_NORMALIZE (WT5_0+2.0*(WT5_1+WT5_2))

float4 PS_Blur_Horizontal_5tap(VS_OUTPUT_BLUR IN) : COLOR
{
    float4 OutCol = tex2D(GlowSamp1, IN.TexCoord0) * (WT5_1/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp1, IN.TexCoord1) * (WT5_2/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp1, IN.TexCoord2) * (WT5_0/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp1, IN.TexCoord3) * (WT5_1/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp1, IN.TexCoord4) * (WT5_2/WT5_NORMALIZE);
    return OutCol;
}

float4 PS_Blur_Vertical_5tap(VS_OUTPUT_BLUR IN) : COLOR
{
    float4 OutCol = tex2D(GlowSamp2, IN.TexCoord0) * (WT5_1/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp2, IN.TexCoord1) * (WT5_2/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp2, IN.TexCoord2) * (WT5_0/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp2, IN.TexCoord3) * (WT5_1/WT5_NORMALIZE);
    OutCol += tex2D(GlowSamp2, IN.TexCoord4) * (WT5_2/WT5_NORMALIZE);
    return float4(Glowness.xyz*OutCol.xyz, Glowness.w);
}

// add glow on top of model

```

```

float4 PS_GlowPass(VS_OUTPUT IN) : COLOR
{
    float4 tex = tex2D(GlowSamp1, float2(IN.TexCoord0.x, IN.TexCoord0.y));
    return tex;
}
technique Glow_5Tap
{
    pass GlowSourcePass
    {
        cullmode = none;
        ZEnable = false;
        ZWriteEnable = false;
        AlphaBlendEnable = false;
        AlphaTestEnable = false;
        FogEnable = False;
        VertexShader = compile vs_2_0 VS_GlowSource();
        PixelShader = compile ps_2_0 PS_GlowSource();
    }
    pass BlurGlowBuffer_Horz
    {
        cullmode = none;
        ZEnable = false;
        ZWriteEnable = false;
        AlphaBlendEnable = false;
        AlphaTestEnable = false;
        FogEnable = False;
        VertexShader = compile vs_2_0 VS_Quad_Horizontal_5tap();
        PixelShader = compile ps_2_0 PS_BluR_Horizontal_5tap();
    }
    pass BlurGlowBuffer_Vert
    {
        cullmode = none;
        ZEnable = false;
        ZWriteEnable = false;
        AlphaBlendEnable = true;
        SrcBlend = one;
        DestBlend = zero;
        AlphaTestEnable = false;
        FogEnable = False;
        VertexShader = compile vs_2_0 VS_Quad_Vertical_5tap();
        PixelShader = compile ps_2_0 PS_BluR_Vertical_5tap();
    }
    pass GlowPass
    {
        cullmode = none;
        ZEnable = false;
        ZWriteEnable = false;
        AlphaBlendEnable = true;
        SrcBlend = one;
        DestBlend = SRCALPHA;
        AlphaTestEnable = false;
        FogEnable = False;
        VertexShader = compile vs_1_1 VS_Quad();
        PixelShader = compile ps_2_0 PS_GlowPass();
    }
}

```

11.7.2 Discussion

Full screen glow effect can also be used for some other effects in the game engine, such as the under water effect. When the camera is under water, we can turn on the full screen effect, and adjust the blending factor so that the blurred image becomes dominant in the final image. Figure 11.15 shows the result of the glow effect which makes the underwater scene more convincing.

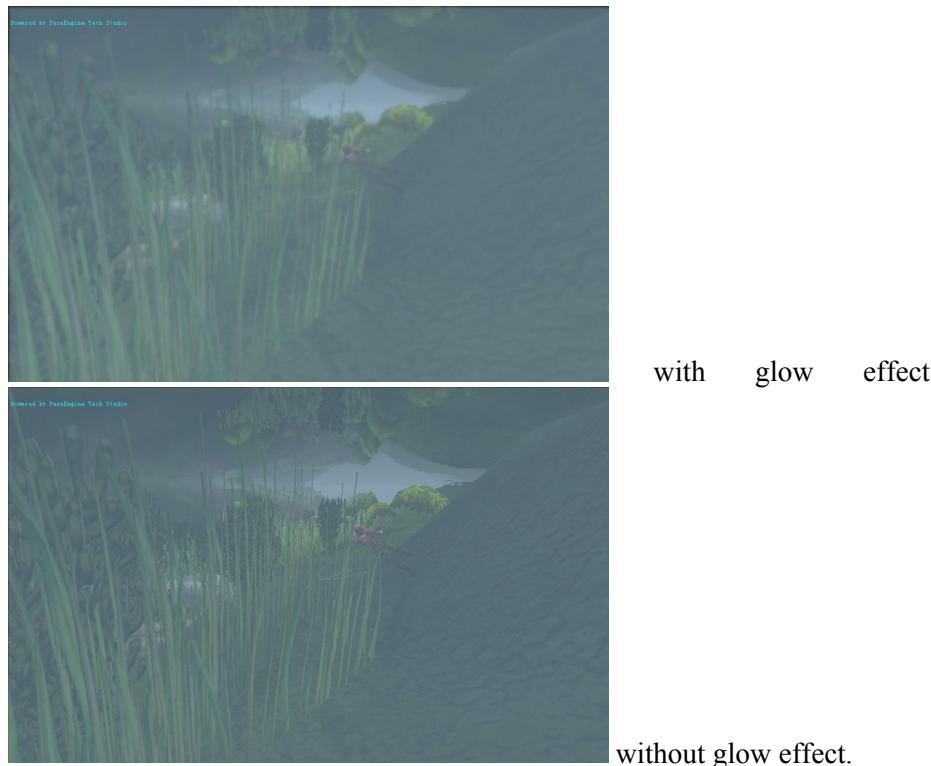


Figure 11.15 Under water effect using full screen glow

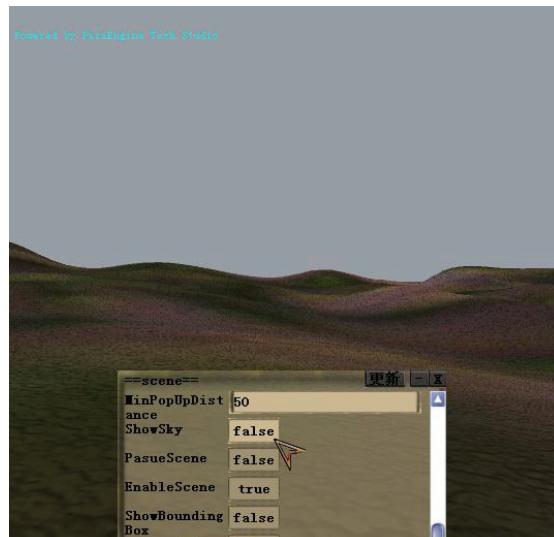
11.8 Fog Effect

Fog effect is mainly used for outdoor scenes to fade objects in to the background color. Fog effect is only two lines of render state codes in fixed function pipeline and just a few lines of code in HLSL shader program if we are using programmable pipeline. The focus of this section, however, is on how to make fog looks right in a game scene, which goes beyond the scope of implementing fog effect itself. For example, the following situations must be dealt with careful in order to make the scene stay in harmony with the global fog effect. (1) Fog and the background sky (2) Fog with global sun lighting (3) Fog underwater (4) Fog with large mesh objects.

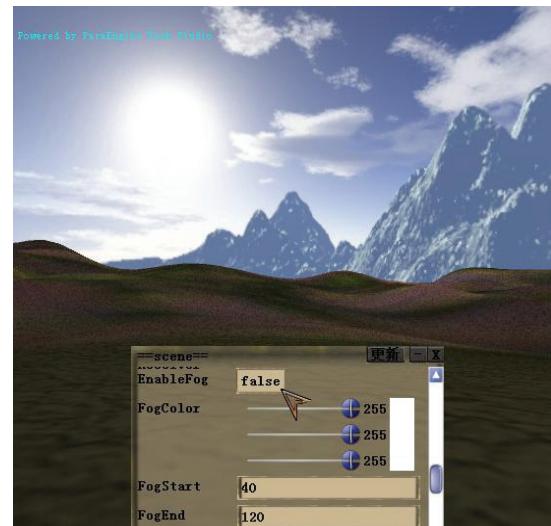
11.8.1 Fog and Sky

In a game scene, the distant fog color must blend into the background sky color. In this section, we will describe a simple way to blend distant fog color with the sky background. The result can be previewed in Figure 11.16.

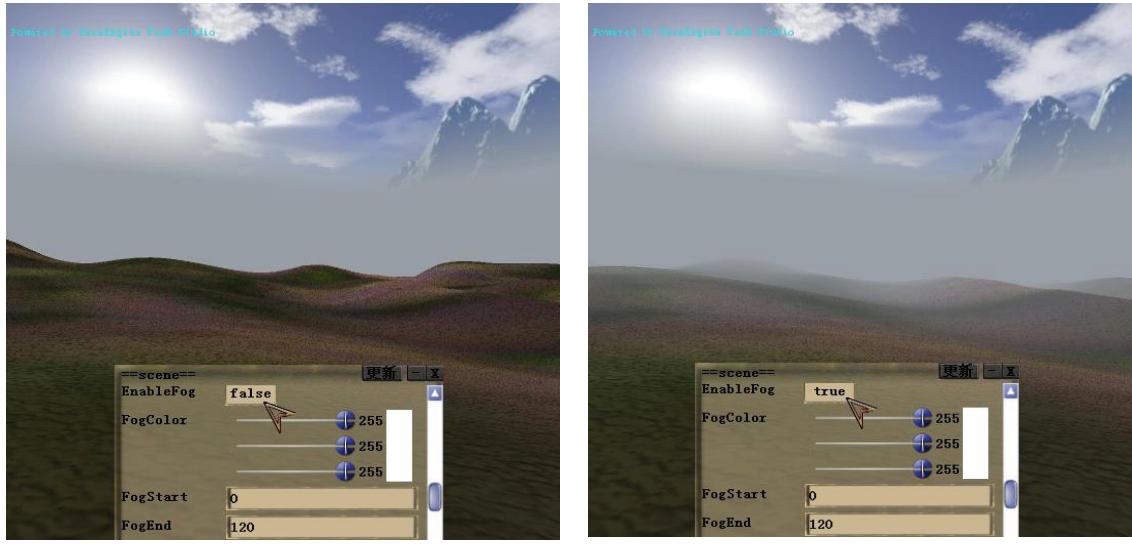
In a game engine, the sky background is rendered first as an ordinary mesh object with Z-buffer disabled and origin fixed at the current camera eye position. The sky mesh object is usually a box, a dome or a plane, depending on the game requirement. The size of the sky mesh is arbitrary (unit size is fine), but since it moves with the camera, it will give the illusion of a sky background in the infinite distance. After rendering the sky, we retrieve the current distant fog color and blend it to the sky background. The blending factor should be degrading in the vertical direction until the fog color fully fades into the background sky color.



No fog, No sky



No fog, No fog blend on sky



No fog, Sky with distant fog blend

Fog and Sky

Figure 11.16 Fog and Sky using Programmable Pipeline

11.8.1.1 Programmable Pipeline Implementation

This effect can be done very easily in shader. The following code shows the sky shader. In this shader, per-pixel fog is used to paint the fog color on the sky model. Yet, instead of using the camera to vertex distance to generate the fog factor, we will use the y (up) component of the sky vertex in object space to generate it.

```

Interpolants vertexShader( float4 Pos : POSITION,
                           float3 Norm : NORMAL,
                           float2 Tex : TEXCOORD0)
{
    Interpolants o = (Interpolants)0;
    // screen space position
    o.positionSS = mul(Pos, mWorldViewProj);
    o.colorDiffuse = colorAmbient+colorDiffuse*sun_color*g_skycolorfactor;
    o.tex.xy = Tex;
    o.tex.z = Pos.y; // y component in object space
    return o;
}
half CalcFogFactor( half d )
{
    half fogCoeff = 0;
    fogCoeff = (d - g_fogParam.x)/g_fogParam.y;
    return saturate( fogCoeff);
}

half4 pixelShader(Interpolants i) : COLOR
{
    half4 o;
    half4 normalColor = tex2D(tex0Sampler, i.tex.xy);
    normalColor.xyz = normalColor.xyz*i.colorDiffuse;

    if(g_bEnableFog)
    {
        //calculate the fog factor

```

```

half fog = CalcFogFactor(i.tex.z);
o.xyz = lerp(g_fogColor.xyz, normalColor.xyz, fog);
o.w = 1.f;/lerp(normalColor.w, 0, fog);
}
else
{
    o = normalColor;
}
return o;
}
technique SkyMesh
{
    pass P0
    {
        // shaders
        VertexShader = compile vs_2_a vertexShader();
        PixelShader = compile ps_2_a pixelShader();

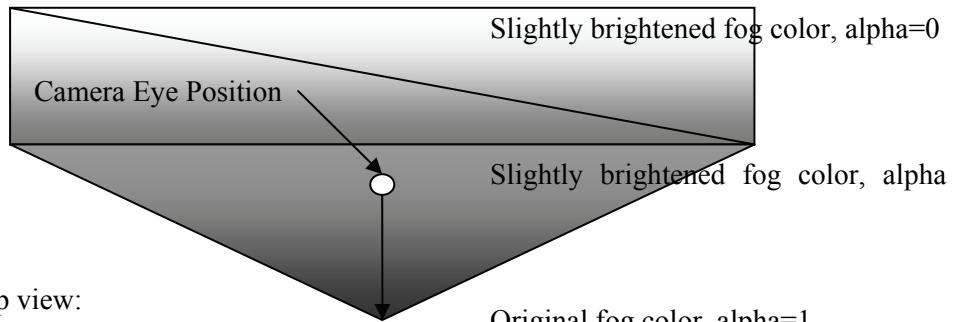
        FogEnable = false;
    }
}

```

11.8.1.2 Fixed Function Pipelined Implementation

If we are building the sky mesh procedurally in the game engine, we can achieve the same effect by encoding the fog factor in the alpha component of vertex color and blend the fog color using this alpha. However, if we are dealing with arbitrary and animated mesh, this method does not work. But we can still use the following method to procedurally build a fog mesh, render and then blend it to the sky background. This fog mesh is shown in Figure 11.17. Triangular strips are used to build the four sides of the fog box, the vertex color at its base has alpha 1.0 which is the distant fog color; whereas the color on its top has alpha 0.0 which will completely blend into the static sky mesh background. Triangular fans are used to build the bottom planes. The center vertex is moved downward for some distance to avoid collision with the camera eye. To make things looks real, the fog color for blending at the far end is brightened by some small factor say 1.1, so that viewers will be able to distinguish the silhouette of an appearing distant object. In case, a reviewer is looking downward from a mountain, the center point of the triangular fan has the exact color of the fog.

Front view:



Top view:

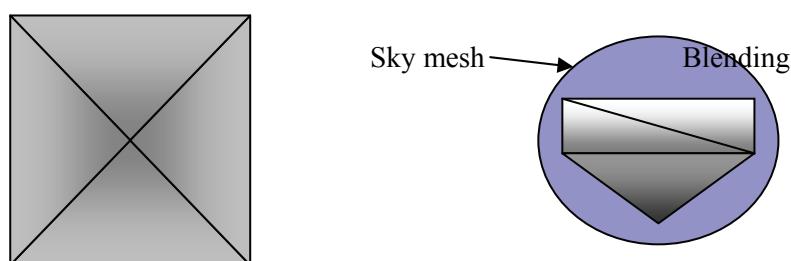


Figure 11.17 Fog color blending

Figure 11.18 shows the implementation result in ParaEngine. We can observe how textures on the sky mesh, the fog, the terrain and other 3D objects coexist in a single scene.



Figure 11.18 Fog implementation result

11.8.2 Fog with Global Sun Lighting

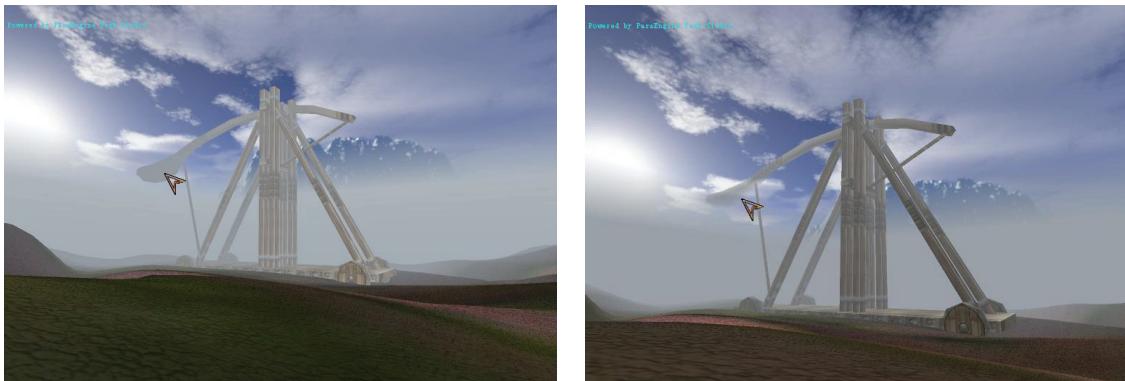
The color of the fog may need to change according to the time of day. In other word, it should change with the global sun lighting parameters. This can be done but using the dot product of the original fog color and a fog factor, which is adjusted by the ambient color of the scene.

11.8.3 Fog underwater

When the camera is under water, we will see a blurred and dimmed underwater world. In the Ocean chapter, we used a full screen quad to dim the entire screen. The color of this quad should be close to the color of the fog to make the underwater effect looks natural. See Figure 11.15.

11.8.4 Fog with Large Mesh Objects

Large or medium sized mesh object can not fully disappear into the distant fog color on the sky mesh. As shown in Figure 11.19, the left image leaves an ugly silhouette filled with distant fog color, when the mesh is out of the camera view frustum or culled by the rendering pipeline when still inside the view frustum, its silhouette will suddenly disappear, which makes very abrupt changes to the graphics. The right image shows a simple remedy to this fog side effect.



Ugly silhouette filled with distant fog color

Nicely blended into the background

Figure 11.19 Fog with large mesh object

In short, it blends the color of the mesh in to the background according to its distance to the camera on a per-pixel basis. This blending factor can be a simple function of the fog factor, as shown in the following pixel shader code snippet.

```
//calculate the fog and blending factor, o is the output color
half fog = i.tex.z;
o.xyz = lerp(normalColor.xyz, g_fogColor.xyz, fog);
half BlendingFactor = saturate( (fog-0.8)*16 );
o.w = lerp(normalColor.w, 0, BlendingFactor);
```

11.9 Conclusion

In this chapter, we have shown several common effect techniques which make the 3D world more realistic. With programmable pipeline, we can now use and play with more special effects in a game than ever before. There are improved technologies that keep emerging every month. We highly advise you to stay connected with the online community, such as the developer websites maintained by NVIDIA and ATI. Another advice to lone wolf engine programmers is that there is actually no need to put too much effect to it until the release time,

since there are tons of other things to care about in order to make a game engine usable by others.

Chapter 12 Character and Animation

One of the most difficult components in a game engine is character animation. Characters are the most eye-catching and interactive elements in a game. For example, we would like a character's appearance to be customizable; we would like to control the body parts of a character separately without compromising the natural movement of the character; and we would like the character to act autonomously and naturally according to very few external commands and complex physical environment, etc. None of these things are simple to implement, and they usually require taking both rendering and simulation in to account.

Chapter 12, Chapter 13 and Chapter 14 all talk about characters in a game engine. In this chapter, we will explain the basics of a character animation system, help developers to build up a basic character animation framework, and give the implementation clues to add today's popular animation features.

12.1 Foundation

12.1.1 Background of Motion Synthesis

When animating a character, there are three kinds of animations which are usually dealt with separately in a game engine system: (1) local animation, which deals with the motion of its major skeleton (including its global speed), (2) global animation, which deals with the position and orientation of the character in the scene, (3) add-on animation, which includes facial animation and physically simulated animation of the hair, cloth, smoke, etc. This chapter mainly deals with the local animation. Local animation is usually affected by the status of the character (such as a goal in its mind) and its perceptible vicinity (such as terrain, water, sounds, etc).

The motion of a certain human character can be formulated by a set of independent functions of time, i.e. $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_0, +\infty]\}$. These functions or variables typically control over 15 movable body parts arranged hierarchically, which together form a parameter space of possible *configurations* or poses. For a typical animated human character, the dimension of the configuration is round 50, excluding degrees of freedom in the face and fingers. Given one such configuration at a specified time, it is possible to render it at real-time with the support of current graphic hardware. Hence, the problem of human animation is reduced to:

Given $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_0, T_c]\}$ and the environment W (or workspace in robotics),

Compute $\{f_n(t) | n = 1, 2, \dots, N; t \in [T_c, T_c + \Delta T]\}$ which should be a realistic human animation.

The most commonly used motion generation technique is to simply play back previously stored motion clips (or short sequences of $\{f_n(t)\}$). The clips may be key-framed or motion captured, which are used later to animate a character. Real-time animation is constructed by blending the end of one motion clip to the start of the next one. To add flexibility, joint trajectories are interpolated or extrapolated in the time domain. In practice, however, they are only applied to situations involving minor changes to the original clips. Significant changes

typically lead to unrealistic or invalid motions. In fact, over 90% percent character animations in 3D computer games today are generated by motion blending from a bank of pre-made animation sequences. We will talk about primarily this method in this chapter.

Alternatively, flexibility can be gained by adopting kinematic models that use exact, analytic equations to quickly generate motions in a parameterized fashion. Forward and inverse kinematic models have been designed for synthesizing walking motions for human figures. There are also several sophisticated combined methods to animate human figures, which generate natural and wise motions (also motion path planning) in a complex environment. Motions generated from these methods exhibit varying degrees of realism and flexibility. However, most of these works are at research level, but we believe that character animation in computer games will be even more intelligent than that in the near future.

12.1.2 Introduction to Skeletal Animation

Skeletal Animation is a technique used to pose character models. A skeleton, which is a hierarchy of bones, is embedded in, and attached to a character model. Once the skeleton is attached, the character model becomes the skin. Posing the skeleton causes the skin to be deformed to match the position of the underlying bones. The beauty of skeletal animation is that (1) all animation data are encoded in bones (2) there is just one piece of mesh (skin), which is shared by all animation sequences (3) there are no seams on the skin after deformation by the bones (4) bone animations are reusable for different skins.

Besides skeletal animation, previous 3D games have used vertex animation and joint animation. In vertex animation, animation data are encoded in each vertex as position keys at each frame. In joint animation, a character mesh is divided in to disjoint body parts and animated individually, which usually create seams or overlaps between disjoint mesh parts during animation. Present day 3D computer games use skeletal animation technique for nearly all kinds of animations. We will only introduce skeletal animation in this chapter. We assume that you already have some non-qualitative knowledge of skeletal animation. And it is even better if you already know how to build skeletal animation models in 3d content creation tools, such as 3dsmax and Maya.

12.1.2.1 Animation Data in Skeletal Animation

There are two sets of data in order to construct a skeletal animation. One is a static mesh, which is a collection of vertices $\{ v_n \}$ in the initial pose. Another is bones, which is a collection of nodes $\{ b_n \}$ containing the node animation (trajectory) as well as a reference to its parent node. The reference in the bones describes the hierarchy of bone structures, as shown in Figure 12.1.

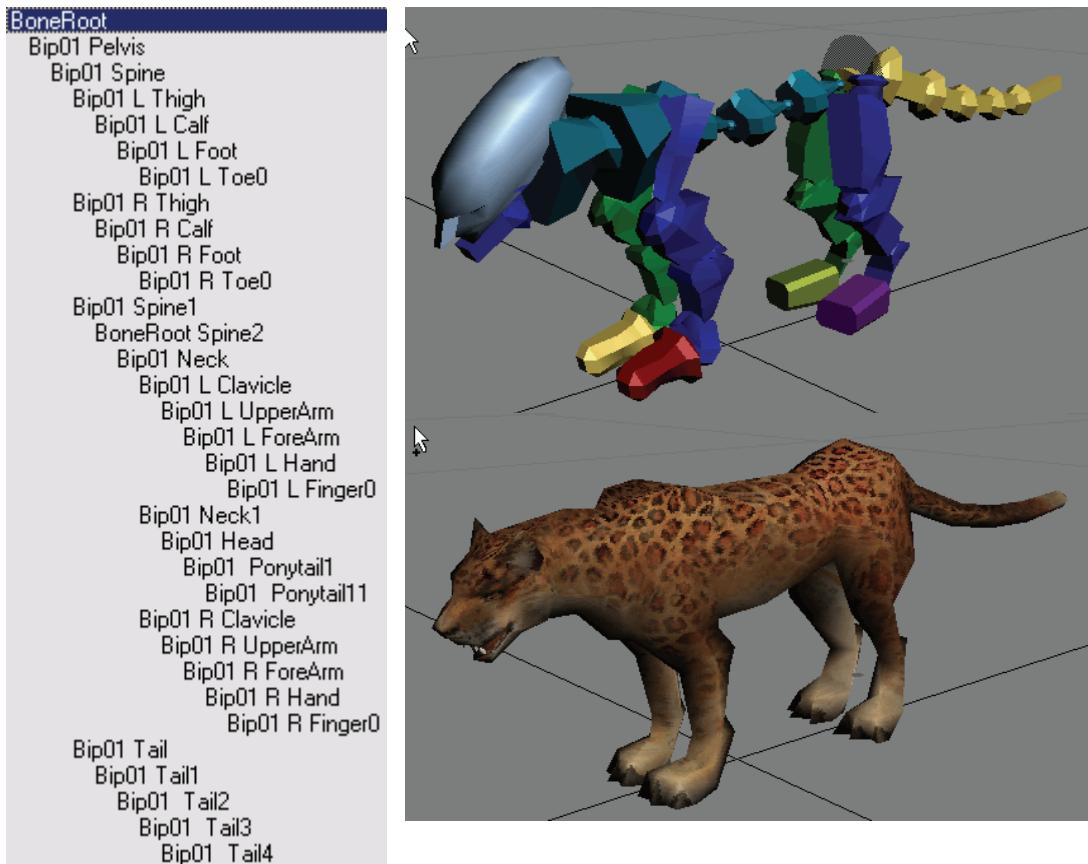


Figure 12.1 Bone, skin and bone hierarchy

The bone's animation data is stored in the form of scale (S), rotation (R) and translation (T) keys relative to its parent bone. We call it SRT keys for short. The animation data used in skeletal animation are usually read from model files, which in turn are exported from 3d content creation tools, such as 3dsmax and Maya. Obtaining the mesh and animation data from its authoring environment is another topic which is discussed in detail in the next chapter. In this chapter, we are primarily concerned with synthesizing and rendering character motions from these pre-made animation data.

Now we are going to define the precise animation data format used in ParaEngine. It should be pointed out that there are still several slightly different ways to store animation data. What will be proposed next is just one of them, which we believe is both efficient and very general for skeletal animation.

A bone is represented by $(\text{pivot}, \text{parent}, \{\text{scale}\}, \{\text{rot}\}, \{\text{trans}\})$, where pivot is a constant 3D vector denoting the pivot point of the bone; parent is the index of the parent bone; $\{\text{scale}\}$ is a set of scale keys for scaling; $\{\text{rot}\}$ is a set of 4D quaternion keys for rotation; $\{\text{trans}\}$ is a set of 3D vector keys for translation. One of the most important function on $\{\text{scale}\}$, $\{\text{rot}\}$, $\{\text{trans}\}$ (or SRT) animation keys, is to query for the key value at a given time t . However, the SRT keys are built from discrete samples from a continuous function of time. In order to reconstruct the function value at any point in the continuous animation time range, we need to interpolate keys. For simplicity, we only consider linear interpolation between two adjacent keys.

$$f(t) = \text{lerp}(f(T_k), f(T_{k+1}), r) = (f(T_k) \times (1.0 - r) + f(T_{k+1}) \times r)$$

where $r = (t - T_k) / (T_{k+1} - T_k)$, and $0 \leq r < 1$

The above interpolation can be used to interpolate the scale and translation keys linearly. For quaternion keys, we use the *slerp* function, which does a spherical linear interpolation between two quaternions $f(T_k), f(T_{k+1})$ by an amount r . See below. The code is given in the code section.

$$f(t) = \text{slerp}(f(T_k), f(T_{k+1}), r) = f(T_k) \frac{\sin((1-r)\Omega)}{\sin \Omega} + f(T_{k+1}) \frac{\sin(\Omega r)}{\sin \Omega}$$

where Ω is angle between $f(T_k)$ and $f(T_{k+1})$, $r = (t - T_k) / (T_{k+1} - T_k)$, and $0 \leq r < 1$

Optionally, each set of keys may contain a field specifying the interpolation method used, which is also exported from the 3d content creation tool. For other interpolation methods, such as HERMITE interpolation, additional information such as the in/out tangent must also be available in each key. So far, linearly interpolation is good enough for most game animations.

12.1.2.2 Calculating Bone Transformation Matrix

For convenience of explanation, we use P , $M_s(t)$, $M_r(t)$, $M_t(t)$ to denote the corresponding homogenous 4x4 matrix of the pivot point translation and the SRT keys at time t for a given bone. (P is translation matrix from world origin to pivot point.)

Let $rM(t)$ denotes the relative transformation matrix, which is a matrix that brings a bone from its parent bone pose to the bone pose at time t . Let $M(t)$ denotes the bone transformation matrix of a bone at time t . The bone transformation matrix is a matrix that brings a bone from its initial bone pose (which is the same pose of the static mesh) to the bone pose at time t .

The relative transformation matrix of the bone at time t can be computed as below. All matrices are row major in this chapter.

$$rM(t) = P^{-1} \times M_s \times M_r \times M_t \times P$$

The bone transformation matrix of a bone at time t can be computed by multiplying the relative bone matrix with the bone transformation matrix of its parent. If we do this recursively, we have:

$$\begin{aligned} M(t) &= rM(t) \times M_p(t) = rM(t) \times rM_p(t) \times M_{pp}(t) = \dots \\ &= rM(t) \times rM_p(t) \times \dots \times M_{root}(t) = \prod rM_i(t), \text{ where root bone } M_{root}(t) = rM_{root}(t) \end{aligned}$$

$rM_p(t)$ and $M_p(t)$ denotes the parent bone's relative transformation matrix and the bone transformation matrix, and so on.

For each bone at time t , we can compute its bone transformation matrix as described above. Hence we can obtain all the bone transformation matrices at time t , and then we can use them to transform mesh vertices to get the mesh pose at time t as described in the following section.

12.1.2.3 Vertex Transformation

For each vertex in the initial mesh pose, the following information is stored.

```
struct ModelVertex {  
    D3DXVECTOR3 pos; // 3D position  
    byte weights[4]; // bone weight of up to four bones  
    byte bones[4]; // bone index of up to four bones  
    D3DXVECTOR3 normal; // vertex normals  
    D3DXVECTOR2 texcoords; // texture coordinates  
    DWORD color0; // always 0,0 if they are unused. Or they can be the second UV set.  
    DWORD color1;  
};
```

What we care about now are three parameters: pos, weights and bones. Pos is the 3D position of the mesh vertex in the initial mesh pose. Bones [4] are the index of up to four bones to which this mesh vertex is bound. Weight[4] is the bounding weights of the corresponding bone. Suppose the bounding weight of the i th bone for the vertex is W_i and the bone transformation matrix of the i th bone at time t is $M_i(t)$, then the world position $v(t)$ of the vertex at time t is given below.

$$v(t) = \sum_{i=0}^n (M_i(t) \times W_i), \text{ where } \sum_{i=0}^n W_i = 1 \text{ and } n \text{ is the total number of bones}$$

Because most W_i is zero for a given vertex, we only keep the index and weight of up to four most important bones per vertex.

For each vertex in the mesh at time t , we use the above method to calculate its position and render the mesh; we do this per frame with a slight increase on time t and it becomes an animation. For character meshes with normals, we need to exclude the position and scale keys when building the transform matrix. Usually we build a separate bone rotation transformation matrix for transforming vertex normals.

So far, we have shown all the steps to animate a mesh on a given animation range using skeletal animation. Next we will show how to motion blend two animation sequences.

12.1.3 Motion Blending

Motion blending is the most frequently used motion synthesis techniques in today's computer games. For a typical game character, artists usually make 10 to 100 animation sequences. They form a bank of motion clips. Motion blending is a technique to concatenate or blend one motion clip to another or to itself. Some common situation of motion blending in computer games are (1) blending the end of one animation sequence to the start of the same animation sequence, such as a looped standing or walking sequence (2) blending one animation sequence at its current play position to the start of another sequence, such as when a character changes from standing to running. In these situations full body motion blending is used; in modern computer game engine, developers may choose to blend only part of the character body to another animation sequence, such as separating the upper body from the lower body. In the latter case, we can achieve more versatile animations, such as walking while waving hands, etc.

The basic implementation of motion blending is to compute two set of mesh poses ($\{v1_n\}$ and $\{v2_n\}$) and use weights to (linearly) interpolate between the two poses. In other

words, for each vertex, let v_1 and v_2 denotes its two poses at two different times, and let w denotes the blending weight (a scalar). The final blended vertex v is computed as below.

$$v_{blended} = v_1 \times (1.0 - w) + v_2 \times w$$

Because this is a linear transform, the result is equivalent to blend v_1 and v_2 's bone transformation matrix. In other words, for each bone, we calculate the blended bone transformation matrix $M_{blended}$.

$$\begin{aligned} v_{blended} &= (v \times M_1) \times (1.0 - w) + (v \times M_2) \times w \\ &= v \times \{M_1 \times (1.0 - w) + M_2 \times w\} \\ &= v \times M_{blended} \end{aligned}$$

we have $M_{blended} = M_1 \times (1.0 - w) + M_2 \times w$

One way of choosing blending poses and their blending weight is given below. Suppose at time T , we want to change animation from animation sequence 1 to sequence 2 and the blending time is N seconds (such as $N=0.3$). In such situation, the mesh pose in sequence 1 at time T is used as the first blending pose in the blending operation for the next N seconds and the poses of sequence 2 during the time $[T, T+N]$ are used as the second pose in the blending operation. The blending weight w during the time $[T, T+N]$ linearly changes from 1 to 0. To use this blending method, the last played frame in sequence 1 is blended with sequence 2 until the blending factor becomes 0. See Figure 12.2. The same rule applies when sequence 1 and 2 are the same sequence.

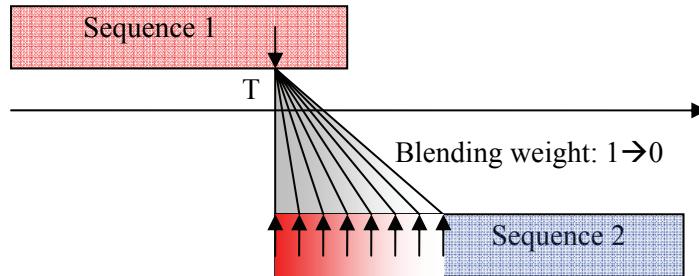


Figure 12.2 Blending poses and weight choice

Figure 12.3 shows the implementation result in ParaEngine. In the figure, the animal character's standing animation is blended to the running animation sequence, as the user commands it to run. The animation snap shot is taken at very short interval.



Pose1:Standing animation

$t=T$



Pose1:Standing animation

$t=T$

Pose2:Running animation

$t=T+0.1s$

Blending Weight =0.66



Pose1:Standing animation

$t=T$

Pose2:Running animation

$t=T+0.2s$

Blending Weight =0.33



Pose1:Standing animation

$t=T$

Pose2:Running animation

$t=T+0.3s$

Blending Weight =0

Figure 12.3 Motion Blending Sample

12.1.4 Character State Manager

With motion blending, we are able to play a looped animation or smoothly change animation from one animation sequence to another. But we have not discussed the module which controls when, and how motion blending should be applied on a given character. A simple animation system can directly map high-level character commands, such as walk, stop, attack, etc to motion blending commands in the animation system. This works if there are only a few animation sequences and a few sequence combinations; yet, in an action RPG game, a character may have over a hundred of short animation sequences. A single command may trigger a series of animation sequences to be played and blended according to the state of the character. For example, consider the scenario when a character jumps down from a high land, safely landed, and then runs in to a water pool, it swims across the water and climbs up to the land. During this process, the user may just be holding the forward action key, yet the character must switch between various animation sequences in order to perform these actions.

To achieve this functionality, we use another logic layer called character state manager to manage motion blending in the animation system. The character state manager is a variant *finite state machine* simulating the low level behavior logics of a character.

12.1.4.1 Finite State Machine

A finite state machine (FSM) is a model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machine), one or more states designated as accepting states (recognizer), etc.

12.1.4.2 Character State Manager

In ParaEngine, we use a state stack and a number of global states to present the state of the character. The following state may appear in the state stack, such as:

```
enum BipedState{
    STATE_MOVING = 0,
    STATE_WALK_FORWARD,
    STATE_RUN_FORWARD,
    STATE_WALK_LEFT,
    STATE_WALK_RIGHT,
    STATE_WALK_BACKWORD,
    STATE_WALK_POINT,
    STATE_SWIM_FORWARD,
    STATE_SWIM_LEFT,
    STATE_SWIM_RIGHT,
    STATE_SWIM_BACKWORD,
    STATE_STANDING = 100, // without speed
    STATE_IN_WATER,      // under or in water
}
```

```

STATE_JUMP_IN_AIR,    // in air
STATE_JUMP_START,
STATE_JUMP_END,
STATE_STAND,
STATE_TURNING,
STATE_ATTACK,
STATE_ATTACK1,
STATE_ATTACK2,
STATE_MOUNT, // mount on target
STATE_DANCE
//... some are ignored
};

```

The global states include things like whether it is running or walking, whether it is mounted, a target position and facing, a timer, and some other miscellaneous states. The character state manager also takes a variety of action symbols as input, such as:

```

enum ActionSymbols{
    S_STANDING = 0,    /// ensure the biped has no speed
    S_IN_WATER,        /// make sure that the biped is in water
    S_ON_FEET,         /// make sure that the biped is on land and on its feet
    POP_ACTION, // pop the current action
    S_STAND,
    S_WALK_FORWARD,
    S_RUN_FORWARD,
    S_WALK_LEFT,
    S_WALK_RIGHT,
    S_WALK_POINT,     // walking to a point
    S_TURNING,
    S_WALK_BACKWORD,
    S_SWIM_FORWARD,
    S_SWIM_LEFT,
    S_SWIM_RIGHT,
    S_SWIM_BACKWORD,
    S_JUMP_START,
    S_JUMP_IN_AIR, // not used.
    S_JUMP_END,
    S_MOUNT,
    S_FALLDOWN,
    S_ATTACK,
    S_ATTACK1,
    S_ATTACK2,
    S_DANCE,
    S_ACTIONKEY, // perform the action in the action key, immediately.
//... some are ignored
    S_NONE
};

```

In the simulation of a single frame, a number of places may feed input in to the character state manager, such as the character physics simulation module, the user input module, the script module, the AI modules, etc. The character state manager has an update() function which is called just before a character model is animated, in which we can hard-code the state transition function. The state transition function will change the character state and produce the proper animation sequence to play next from all input action symbols in that frame; hence when the character model is animated, it knows which animation sequence to play and how to motion blend it. Some code sample can be found in the code section.

12.1.5 Attachment and Custom Appearance

Nowadays, players can customize the appearance of their avatars in computer games, such as changing the hair style, skin color, wearing many different kinds of clothes, handling different weapons, etc. These effects are achieved by a mixture of the following methods

- (1) Dynamically compose the character's model texture from a collection of texture files according to the user selection. Different skin colors, eye colors, etc, are achieved by this method. Different clothes on a character body may also be achieved in this way.
- (2) Hide and show geometry sets in the character model according to user selection. For example, a character model may contain geometry sets for all supported hair styles, while it only shows the one selected by the user at render time. For another example, a character may have a robe geometry which is only shown when the character is wearing a robe.
- (3) Textures of some geometries can be replaced by the one selected by the user. This is similar to the first one; but it does not require dynamically composing texture object, which has some computation overhead.
- (4) Attach external mesh objects to bones of the main character. Weapons, irregular shaped armories, etc are usually achieved in this way.
- (5) Use a different character model. One can not change a male into a female from the above methods; hence, it is better to use a completely different model if we can not customize it.

Managing character appearance might require a light-weight database system, because we need to keep additional information for every attachable mesh object and every character mesh. For example, for each attachable mesh, such as a red robe, we need to know which geometry sets (geosets) on the main character model need to be displayed when the robe is attached and we also need to know if the red robe requires that some replaceable textures on the character body also be changed to some pre-designed textures, so that they match the color of the robe. If there are thousands of attachable models in a game, using a database to store the additional information is a good choice.

In ParaEngine, we use a data provider class called CharacterDB to interface the game engine's character animation system to the character database. For example, the database provider class may expose the following query functions to the animation system.

```
/** Get race ID from name
return true if the record is found in database. */
bool GetRaceIDByName(const string& racename, int& id);

/** get the model ID from model asset file name
return true if the record is found in database. */
bool GetModelIDfromModelFile(const string& sModelFile, int& modelid, int& modeltype);

/** get the replaceable texture group for the specified index.
return true if the model is found.but it does not mean that the skin ID is found.
@param bFound: whether found */
bool GetReplaceTexturesByModelIDAndSkinID(int modelid, int skinIndex, string& sReplaceTexture0,
string& sReplaceTexture1, string& sReplaceTexture2, bool& bFound);

/** get the character's skin textures with a specified section type and section number.
```

```

return true if the record is found in database. */
bool GetCharacterSkins(int race, int gender, int nSectionType, int nSection, int skinColor, string&
sSkinTexture0, string& sSkinTexture1, string& sSkinTexture2);

/** get the hair style geoset.
return true if the record is found in database. */
bool GetFacialHairGeosets(int race, int gender, int hairStyle, int& geoset);

/** get item's model type and model id by its item id.
return true if the record is found in database. */
bool GetModelIDfromItemID(int itemid, int& nItemTypeID, int&nItemModelID);

/** get item display information by its model item id. The display information will describe how the item
(equipment) is displayed on the body of a character.
return true if the record is found in database. */
bool GetModelDisplayInfo(int nItemModelID, int GeosetA, int GeosetB, int GeosetC,
string& skin, string& TexArmUpper, string& TexArmLower, string& TexHands, string&
TexChestUpper, string& TexChestLower, string& TexLegUpper, string& TexLegLower, string& TexFeet);
//... some functions are ignored here

```

12.2 Architecture

In ParaEngine, there are three major classes for character animation. They are CParaXModel, CParaXAnimInstance and CBipedStateManager. They are explained in the following sections separately.

12.2.1 Model File

CParaXModel presents a loaded model file, which contains the model mesh, bones and bone animation data, etc. There is a class called CParaXSerializer which is used to parse a character model file and construct the corresponding CParaXModel object. Two of the most important functions exposed by the CParaXModel class are Animate() and Draw(). They are usually called in pair by each character instance. The Animate function calculate all motion blended bone transformation matrices used by the animation and the Draw function transforms all visible vertices by their related bone matrices to get the right animation pose and renders the mesh. Calculation in the Draw function can be done both in CPU and GPU. When doing the calculation on GPU, bone transformation matrices are uploaded to GPU registers or buffers which are used by the vertex shader.

12.2.2 Animation Instance

There may be many character instances in the scene sharing the same main model file. Each character instance is associated with an animation instance, which contains the information of the custom appearance of the character model (also its attached models if any) as well as the animation frames to play. Whenever the render pipeline ask a character to render itself, it first finds its associated animation instance; the animation instance will update its associated model file object(s) according to its custom appearance and animation parameters, such as motion blending frame numbers; finally the model file object(s)' animate and draw method are called to render the mesh in the requested motion blended pose. Each animation instance class may work with one or several kinds of model files. In ParaEngine, CParaXAnimInstance class is the animation instance class working with both animated and

static models stored in CParaXModel model file. The relationships between these classes are given in Figure 12.4.

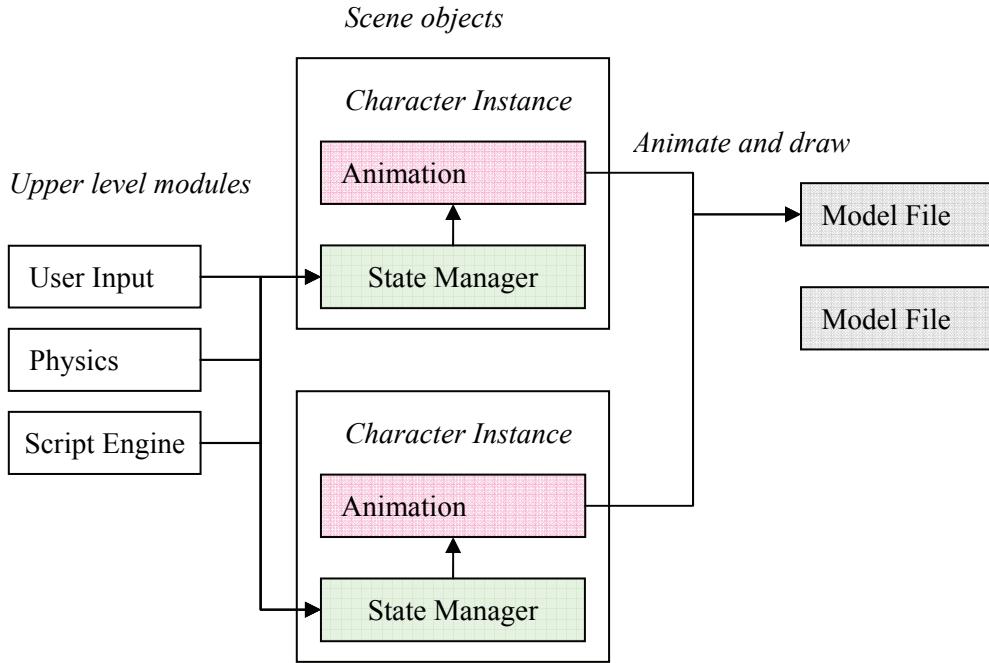


Figure 12.4 Relationships between character animation modules

12.2.3 State Manager

Character state manager is discussed in section 12.1.4.2. CBipedStateManager is a special implementation of a character state manager. It is mainly written for humanoid models. A humanoid model is characterized by some special bone structures in the model files, such as the neck bone, the waist bone, etc and also by some predefined animation sequences in the model file, such as walk, turn, jump, swim, etc. One may develop different state managers for different model types, such as for wheeled vehicle models or ship models, etc.

12.3 Code and Performance Discussion

Some useful code snippets used in this chapter are given.

12.3.1 Spherical Linear Interpolation of Quaternion

Here is the slerp function code for interpolating quaternion animation keys.

```

static const Quaternion slerp(const float r, const Quaternion &from, const Quaternion &to)
{
    float to1[4];
    float omega, cosom, sinom;
    float scale0, scale1;

    cosom = from.x * to.x + from.y * to.y + from.z * to.z
        + from.w * to.w;
  
```

```

// adjust signs (if necessary)
if ( cosom < 0.0 )
{
    cosom = -cosom;

    to1[0] = - to.x;
    to1[1] = - to.y;
    to1[2] = - to.z;
    to1[3] = - to.w;
} else {
    to1[0] = to.x;
    to1[1] = to.y;
    to1[2] = to.z;
    to1[3] = to.w;
}

// calculate coefficients
if ( cosom < 0.9995f )
{
    // standard case (slerp)
    omega = acosf(cosom);
    sinom = sinf(omega);
    scale0 = sinf((1.0f - r) * omega) / sinom;
    scale1 = sinf(r * omega) / sinom;
} else {
    // if "from" and "to" quaternions are very close, interpolate directly
    scale0 = 1.0f - r;
    scale1 = r;
}
// calculate final values
Quaternion res;
res.x = scale0 * from.x + scale1 * to1[0];
res.y = scale0 * from.y + scale1 * to1[1];
res.z = scale0 * from.z + scale1 * to1[2];
res.w = scale0 * from.w + scale1 * to1[3];
return res;
}

```

12.3.2 Retrieving Animation Keys

We can use a template class for storing and retrieving scale, rotation and translation keys. See below.

```

typedef std::pair<int, int> AnimRange;

template <class T>
class AnimatedKey {
public:

    bool used;
    int type, seq;

    std::vector<AnimRange> ranges;
    std::vector<int> times;
    std::vector<T> data;
    // ..method ignored
}

```

Animation keys are stored in indexed time and value arrays, such that a stored key at index N can be retrieved as (times[N], data[N]). Animation sequences are stored in an array of

AnimRange objects. For a given animation sequence index K, ranges[K].first, ranges[K].second are the index of the first and last key index in the time and data array.

Specific animation keys such as translation and rotation keys are instantiated like this:

```
AnimatedKey<D3DXVECTOR3> translationKeys;
AnimatedKey<Quaternion> rotationKeys;
```

In order to retrieve a given key from a given animation sequence at a given time, we can perform a binary search on the time range of the given sequence. The code is given below. Please note that we have shown two versions in the code, the first getValue() function returned the non-motion blended key value, whereas the second one returns a motion blended key value.

```
class AnimatedKey{
public:

    /** this function will return the interpolated animation vector at the specified anim id and frame
    number*/
    T getValue(int anim, int time)
    {
        if (type != INTERPOLATION_TYPE_NONE || data.size() > 1) {
            AnimRange range;

            // obtain a time value and a data range
            if (seq != -1) {
                /// global animation has nothing to do the current animation. Such animation may be
                /// the blinking of eyes and some effects which always loops regardless of the sequence.
                if (globals[seq] == 0)
                    time = 0;
                else
                    time = globalTime % globals[seq];
                range.first = 0;
                range.second = (uint32)(data.size()) - 1;
            } else {
                /// get the range according to the current animation.
                range = ranges[anim];
                // there is a chance that time = times[times.size()-1]
                //time %= times[times.size()-1]; // I think this might not be necessary?
            }

            if (range.first != range.second) {
                size_t pos = range.first; // this can be 0.
                {
                    /** use binary search for the time frame */
                    int nStart = (int)range.first;
                    int nEnd = (int)range.second - 1;
                    while(true)
                    {
                        if(nStart >= nEnd)
                        { // if no item left.
                            pos = nStart;
                            break;
                        }
                        int nMid = (nStart + nEnd) / 2;
                        int startP = (times[nMid]);
                        int endP = (times[nMid + 1]);
                
```

```

        if(startP <= time && time < endP )
        { // if (middle item is target)
            pos = nMid;
            break;
        }
        else if(time < startP )
        { // if (target < middle item)
            nEnd = nMid;
        }
        else if(time >= endP)
        { // if (target >= middle item)
            nStart = nMid+1;
        }
    } // while(nStart<=nEnd)
}
int t1 = times[pos];
int t2 = times[pos+1];
float r = (time-t1)/(float)(t2-t1);

if (type == INTERPOLATION_TYPE_LINEAR) return
interpolate<T>(r,data[pos],data[pos+1]);
else if (type == INTERPOLATION_NONE){
    return data[pos];
}
else {
    // INTERPOLATION_TYPE_HERMITE is only used in cameras
    return interpolateHermite<T>(r,data[pos],data[pos+1],in[pos],out[pos]);
}
} else {
    return data[range.first];
}
} else {
    // default value
    return data[0];
}
}

/**
 * get key value with motion blending with a specified blending frame.
 * @param nCurrentAnim: current animation sequence ID
 * @param currentFrame: an absolute ParaX frame number denoting the current animation frame. It is always within
 * the range of the current animation sequence's start and end frame number.
 * @param nBlendingAnim: the animation sequence with which the current animation should be blended.
 * @param blendingFrame: an absolute ParaX frame number denoting the blending animation frame.
It is always within
 * the range of the blending animation sequence's start and end frame number.
 * @param blendingFactor: by how much the blending frame should be blended with the current frame.
 */
T getValue(int nCurrentAnim, int currentFrame, int nBlendingAnim, int blendingFrame, float
blendingFactor)
{
    if(blendingFactor == 0.0f)
    {
        return getValue(nCurrentAnim, currentFrame);
    }
    else

```

```

    {
        if(blendingFactor == 1.0f)
        {
            return getValue(nBlendingAnim, blendingFrame);
        }
        else
        {
            T v1 = getValue(nCurrentAnim, currentFrame);
            T v2 = getValue(nBlendingAnim, blendingFrame);

            return interpolate<T>(blendingFactor,v1,v2);
        }
    }
}

```

12.3.3 Animation Instance Interface

Animation Instance keeps information about the custom appearance and animation parameters for a character instance. It contains a member of CharModelInstance, which contains the custom appearance attributes and body attachments. This member is only valid if the animation instance is for customizable character model, other than a non-customizable model.

```

Class CparaXAnimInstance :
    public CAnimInstanceBase
{
public:
    CParaXAnimInstance(void);
    ~CParaXAnimInstance(void);

public:

    /** defines the model type that this instance holds*/
    enum ModelType{
        /** the character model is a fully customizable model for human shaped animation model
         * one can change the skin, attachments, clothings, hair and facial styles, etc */
        CharacterModel=0,
        /** Fixed model is a model which is not customizable, but can contain a list of animations to play
         * such as a flying flag, etc.*/
        FixedModel=1
    };

    /** model animation information */
public:
    /** current animation index, this is different from sequence ID */
    int nCurrentAnim;
    /** an absolute ParaX frame number denoting the current animation frame. It is always within
     * the range of the current animation sequence's start and end frame number. */
    int currentFrame;
    /** the next animation index.if it is -1, the next animation will
     * depends on the loop property of the current sequenc; otherwise, the animation specified
     * by the ID will be played next, after the current animation reached the end. */
    int nNextAnim;

    /** the animation sequence with which the current animation should be blended. */
    int nBlendingAnim;
    /** an absolute ParaX frame number denoting the blending animation frame. It is always within
     * the range of the blending animation sequence's start and end frame number. */
    int blendingFrame;
}

```

```

/** by how much the blending frame should be blended with the current frame.
 * 1.0 will use solely the blending frame, whereas 0.0 will use only the current frame.
 * [0,1), blendingFrame*(blendingFactor)+(1-blendingFactor)*currentFrame */
float blendingFactor;

private:
    /// the type of the model
    ModelType m_modelType;
    union {
        CharModellInstance * m_pCharModel;
    };
    ParaXEntity * m_pModel;

public:

    /** Get the character model instance. If the model instance is not of character type,
     * then NULL will be returned. But it does not mean that there is not a valid model instance of other
     * types */
    CharModellInstance * GetCharModel();
    /** get the animation model*/
    ParaXEntity* GetAnimModel();
    /** init the animation instance. associate it with the ParaX model*/
    void Init(ParaXEntity * pModel);

    /** refresh the model appearance with current settings. Call this function whenever the model has
    changed its appearance. */
    void RefreshModel();

    /** Render the model with its current settings.*/
    virtual HRESULT Draw( SceneState * sceneState, const D3DXMATRIX* mxWorld);

    /** shadow volume */
    virtual void BuildShadowVolume(ShadowVolume * pShadowVolume, LightParams* pLight,
D3DXMATRIX* mxWorld);

    /** Advance the animation by a time delta in second */
    virtual void AdvanceTime( double dTimeDelta );
    /** Get the specified attachment matrix of the current model.
     * this is usually for getting the mount point on a certain model, such as horses.
     * @return: NULL if not successful, otherwise it is pOut.
     */
    D3DXMATRIX* GetAttachmentMatrix(D3DXMATRIX* pOut, int nAttachmentID);

    virtual void LoadAnimation(const char * sName, float * fSpeed,bool bAppend = false);
    virtual void LoadAnimation(int nAnimID, float * fSpeed,bool bAppend = false);

// ... some functions are ignored.
};

```

12.3.4 Character State Manager

The character state manager is a variant *finite state machine* simulating the low level behaviors of a character. Following is some state functions used in our CBipedStateManager class.

```

class CBipedStateManager
{

```

```

public:
private:
    /// the state memory
    list <BipedState> m_memory;
    /// the biped object that is associated with this state manager.
    CBipedObject* m_pBiped;
    /// state timer, it is used for state transition timing.
    /// for example, one can define the time from state1 to state2.
    float m_fTimer;
    /** last animation state, which is the animation played in the last frame.
     * this is used for AI or other modules to check for the current biped animations.*/
    BipedState m_nLastAnimState;
    /** true if character uses walk as the default moving animation.otherwise it uses running.
     * default value is running.*/
    bool m_bWalkOrRun:1;
    /** whether the biped's state is being recorded to the movie controllers. Default value is false. */
    bool m_bRecording:1;
    /** whether the biped is mounted on another object. */
    bool m_bIsMounted:1;

    /* User data */
    D3DXVECTOR3 m_vPos; // for position user data
    float m_fAngleDelta; // for angle user data
protected:
    /** remove all occurrence of a given state */
    void RemoveState(BipedState s);
    /** make sure that the memory has space left for one more state*/
    void CheckMemory();

    /** replace the current state with the one in the input*/
    void ReplaceState(BipedState s);
    /** add a new state to state memory */
    void PushState(BipedState s);
    /** all occurrence of the same state will be removed from the memory before this state is pushed to
the memory*/
    void PushUniqueState(BipedState s);
    /** prepend a new state to state memory*/
    void PrependState(BipedState s);
    /** all occurrence of the same state will be removed from the memory before this state is
prepended to the memory*/
    void PrependUniqueState(BipedState s);
    /** ensure that state s is unique in the memory. If there is one than one such state, it will be
deleted.*/
    void SetUniqueState(BipedState s);
public:
    /** whether the object is mounted on another object. */
    bool IsMounted();
    /** set mount state */
    void SetMounted(bool bIsMounted);
    /** return true if character uses walk as the default moving animation.otherwise it uses running.*/
    bool WalkingOrRunning();
    /** Set default moving style
     *@param bWalk: true if character uses walk as the default moving animation.otherwise it uses
running.*/
    void SetWalkOrRun(bool bWalk);

    /** last animation state, which is the animation played in the last frame.
     * this is used for AI or other modules to check for the current biped animations.*/
    BipedState GetLastAnimState();

```

```

/** Find a specified state in memory.
 * @param s: the state to be found.
 * @return: If the state is not found, a negative value(-1) will be returned; otherwise the current
index of the first occurrence of the state is returned.*/
int FindStateInMemory(BipedState s);

/** get the last biped state */
BipedState GetLastState();
/** get the first biped state */
BipedState GetFirstState();

static bool IsStandingState(BipedState s);
static bool IsMovingState(BipedState s);
/** whether the memory contains at least one moving state.*/
bool HasMovingState();
/// whether the biped is swimming
bool IsSwimming();
/// whether the biped is flying
bool IsFlying();

CBipedObject* GetBiped();
void SetBiped(CBipedObject* pBiped);

/** read an action symbol, and let the manager determine which states it should go to.
* this function will not perform any concrete actions on the biped objects. Call
* Update() method to perform these actions on the associated biped after adding series of
* action inputs to the state manager. The recommended order of calling is
* StateManager::Update()->Environment Simulator->IO:{StateManager::AddAction}
* @param nAct: the action
* @param data: the data specifying more details about the action. This value default to NULL
* if nAct is S_ACTIONKEY, then pData is const ActionKey*
* if nAct is S_WALK_POINT, then pData is NULL or 1, specifying whether to use angle.
* @return: the current state is returned. */
BipedState AddAction(ActionSymbols nAct, const void* pData=NULL);

/** update the associated biped according to the memory or state of the manager.
* this function is automatically by each active biped during each frame move.
* it may sometimes change the memory of state manager. But in most cases, it just carries out
* actions according to the current state.
* @param fTimeDelta: the time elapsed since the last time this function is called. Units in
seconds.*/
void Update(float fTimeDelta);
//.. several miscellaneous functions are ignored
}

```

12.4 Summary and Outlook

The character animation system discussed in this chapter can be extended to synthesis more intelligent and realistic character motions. Character animation is an active research field and its development never slows down in the past decades.

When looking forward in this domain, we are always relating it to us human beings. I.e. how we learn to control the motion of our body by learning, observing, and practicing in the physical environment and how we are able to generate motions to achieve our mental goals with little mental and physical (energy) cost. There is still a long way to go to make virtual

characters in the virtual world act like ourselves. But whatever advancement people make, you are likely to have the first experience of it in a computer game.

In the next chapter, we will see how to generate animation data from 3d content creation tools.

Chapter 13 Writing Model Exporters

A game engine, though it could be, is not an all-in-one game development platform. It still depends on a number of external third party tools to create the content used in a game title. Famous 3D content creation tools are 3dsmax, Maya, etc. One of the most important external tools that virtually every game engine needs to provide is the model exporter. It exports 3D models, textures, materials, and animation data from these widely used professional tools to model files known to the game engine. In this chapter, we will take 3dsmax as an example and show the steps of writing a model exporter. Optionally, a game engine may also develop tools to export a complete 3D scene, including terrain, referenced 3D models, lights, paths or even game specific logics from external content creation tools to the game engine.

13.1 Foundation

Professional 3D content creation tools like 3dsmax and Maya have exposed API through both the C++ programming interface and their built-in scripting system. These SDK API enables developers to write plug-in programs to extend their functionalities. Models and animations used by a game engine are first constructed in these tool environments and then exported through plug-in programs to game files known to the game engine.

The relationship between a game engine and 3D content creation tools usually defines the major game production line. And it may vary from game to game even for the same game engine. Before a game is made, game designers and engine designer (or tool developers) must jointly decide which portion of the game content should be built in third-party tools, and which are to be developed and tested using in-game tools. But whatever their relationship, its design goal is to make the game production line more efficient for a specific game and for a specific team.

Model exporter is the most common plug-in program which is almost always developed no matter what the game is. In this chapter, we will take 3dsmax as an example and show the steps of writing a model exporter. The concept used in writing 3dsmax based exporter can be mapped to other 3D tool platforms as well. Developing model exporter program is also a good starting point for learning the SDK API in 3dsmax.

13.1.1 Terminologies

Before we start, let us get familiar with a few important terminologies in this domain. They can be easily confused when writing the model exporter. We assume that you know how to use a 3d modeling tool, such as 3dsmax, to create skinned or biped animation, and that you know what a skin modifier is.

13.1.1.1 Row major and Column major Matrix

If this is the first time you ever worked with other 3D software SDK besides your own, you may run in to trouble of managing multiple definitions of matrix and coordinate systems. If you are working with DirectX, you are primarily working with row major matrix and in left-handed coordinate system. OpenGL and 3dsmax by default uses column major matrix. When

writing model exporters, we had better adhere to a single coordinate system and math operators. In this book, we use row major matrix and math. Hence if v is a 4D vector, and m is a 4*4 matrix, we will assume that they are all row major and we use the following math to transform the vector. This math convention is also adopted by DirectX.

$$v' = v \times m$$

If you are using 3dsmax's Data eXchange Interface (3DXI) or originally called IGame interface API before version 7, you can count on 3dsmax to automatically convert the coordinate system for you. You can tell the 3dsmax to use DirectX compatible coordinate system, and then everything you get from the 3dsmax API will be row major, left-handed with the y component pointing upwards. Right now, some helper mathematical functions in 3dsmax still use its internal coordinate format; hence one still need to pay attention and convert these things manually in a few places.

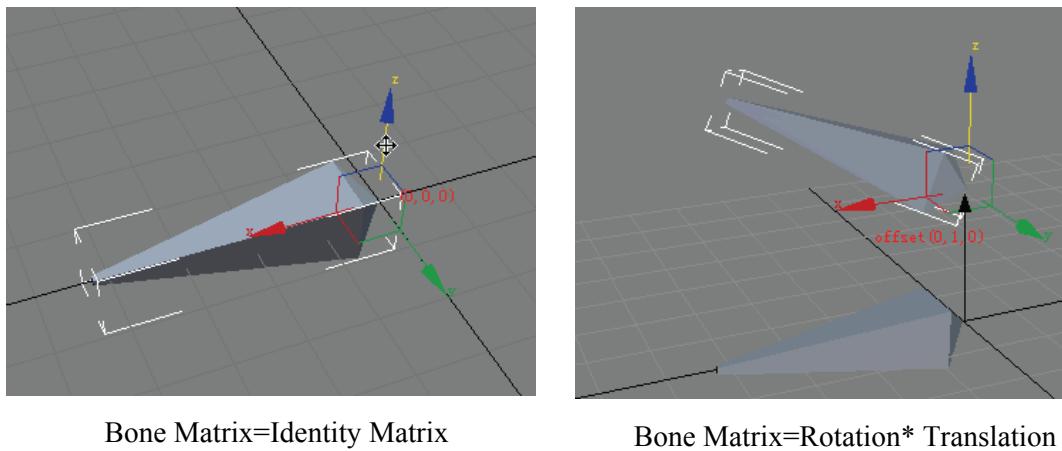
13.1.1.2 Initial Pose

Initial Pose refers to the static mesh pose when the skin modifier is applied. It is more precise to say that it is a snapshot of the static mesh just before the skin modifier is applied. This is different from the animated mesh pose at Frame 0, because the mesh at Frame 0 is already transformed by the skin or biped modifier in 3dsmax. Now suppose we have three sequences of character animations stored in three separate files: one for standing, one for walking, and the other is for running. For a game engine using skeletal (or skinned) animation, it is required that all characters in these three files share the same skeleton and *initial pose*.

There are various ways to make skeletal animation in 3dsmax, extracting this initial pose using the standard API can be difficult. Fortunately, 3dsmax exposes a very handy function called GetInitialPose() through 3DXI. The mesh returned from this function can be used as the shared mesh for all skinned animation sequences.

13.1.1.3 Bone's World Transform Matrix

The effect of a bone is mathematically equivalent to a special 4*4 matrix M . In 3dsmax, this matrix is called the bone's world transformation matrix. To graphically represent and manipulate a bone, 3dsmax uses a bone-like mesh whose origin is at (0,0,0). It then transforms the bone mesh using the bone's world transform matrix M . Now the transformed mesh becomes the graphical presentation of the bone matrix M . For example, an identity matrix draws a bone at the world origin, whereas a translation matrix draws the bone mesh offset by the mount in the matrix's last row. See Figure 13.1. To modify bones, artists manipulate the bone mesh in 3D space, such as translating, rotating and scaling it, the result will be propagated to the bone matrix.



Bone Matrix=Identity Matrix

Bone Matrix=Rotation* Translation

Figure 13.1 Bone Matrix and its graphical presentation in 3dsmax

13.1.1.4 Initial Bone Matrix

Initial bone matrix is a special constant world transformation matrix which brings the bone mesh from its local space (or you can think of it at world origin) to its initial pose in the world space, where it is bound with the *initial pose* of the mesh. The presentation of bones when all of them are transformed by their initial bone transform matrix is called the initial bone pose. It is at the initial bone pose that the character mesh is bound. The snapshot of the mesh when bounded to initial bone pose is called the initial pose of the mesh, as explained in the previous section. See Figure 13.2.

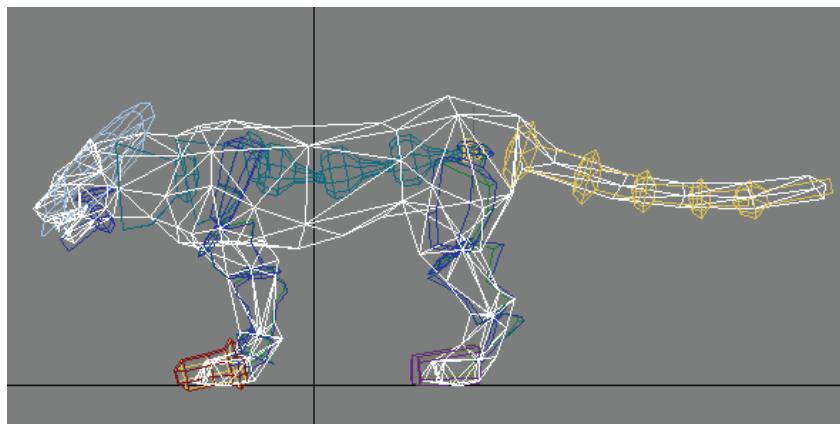


Figure 13.2 Initial bone pose and initial mesh pose

Like initial pose, the initial bone transform matrices of all bones in all animation sequences of a character must be the same. In essence, this ensures that all sequences of a character share the same skeleton. In 3dsmax's 3DXI interface, we can get the initial bone transformation matrix at any time by calling `GetInitBoneTM()` function.

13.1.1.5 Bone's World Transform Matrix at Frame N

In 3dsmax, we can sample the animation data by querying the bone's world transformation matrix at a specific frame within the animation range. Like the initial bone matrix, the world transform matrix at frame N brings a bone from its local space to its bone pose at Frame N.

For example, Figure 13.3 shows the bone pose at frame 0, which is the first frame of the running sequence.

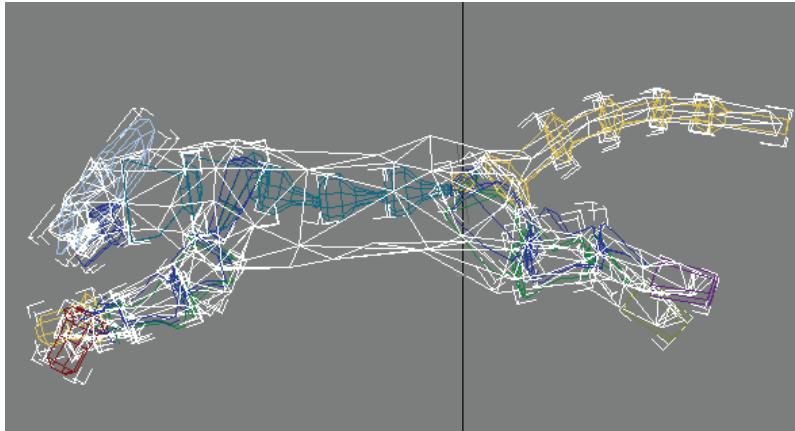


Figure 13.3 Bone pose at frame N

13.1.1.6 Affine Matrix Decomposition

There are basically three kinds of bone manipulations in 3dsmax. They are scaling (S), rotation (R) and translation (T). We call them SRT transformation in short. An artist may apply these operations to a bone any number of times and in any order. Although 3dsmax internally keeps the history of these operations in terms of SRT keys, it is difficult to get them through its API due to the number of ways that a bone may be animated in 3dsmax. However, we can get the bone's world transformation matrix at any frame (time) regardless of the type of keys through the 3DXI API. We will show that, by decomposing a bone's transformation matrix in to SRT keys, we will be able to export interpolable animations to the game engine.

Matrix Interpolation

The reason for a game engine to use SRT keys instead of the bone's transformation matrix itself is for smooth interpolation between two adjacent key frames. In previous chapter, we have seen that animations in a game engine are generated by interpolating the SRT keys separately. You might ask: will it be just good to linearly interpolate the bone's world transformation matrix? Well, the answer lies in how you define a good interpolation. Interestingly, this is a psychological problem concerning how the human mind interpret and predict motions that it observes. Experiments show that the human mind tends to interpret contents on image sequences as solid bodies, which are rotated and/or translated using the shortest path from one image to the next. If the size of symbols on two consecutive images changes, the mind would most likely to interpret it as a scale. In other words, the human mind interprets a sequence of images and predicts the non-existent ones in-between by rules of SRT as well. On the other hand, we will get unnatural motion if we linearly interpolate $M(t)$ and $M(t+1)$, using the 16 components of the 4×4 matrix M . However, if we decompose the matrix in to SRT terms, interpolate them individually, and then reassemble them back to a single matrix, we will obtain more naturally interpolated motion during the time $(t, t+1)$. This is why decomposing a matrix into SRT components is very important in computer graphics. The process is explained below.

Affine Matrix

Three types of matrix are commonly used for 3-D graphics: 3×3 linear, 3×4 affine, and 4×4 homogeneous; similar types with one less column and row are used for 2-D graphics. The homogeneous matrix is most general, as it is able to represent all the transformations required to place and view an object: translation, rotation, scale, shear, and perspective. Shear and perspective transformations are not used when animating bones. All transformations except perspective can be accommodated by an affine matrix, which, in turn, can be considered just a 3×3 linear matrix with a translation column appended. Hence matrix used in 3D animation is 3×4 affine matrix with 12 degrees of freedom.

Affine Matrix Decomposition

Decomposing affine matrix is a computationally extensive iterative process. One can find the details in the paper “Matrix Animation and Polar Decomposition” written by Ken Shoemake, and also in an article in Graphics Gem IV. Fortunately, 3dsmax SDK contains such a function called `decomp_affine()`, which takes a 3dsmax affine matrix and returns its SRT parts. When we say 3dsmax affine matrix, we really mean it. You can not provide your own affine matrix; you have to use the 3dsmax’s presentation of matrix in order to get the correct result. We hope that 3dsmax will release a more public version of this function in its next release. The input and output of this function are related by the following equation. Please pay attention to the order; it is S, R, and then T.

$$mSRT = Ms \times Mr \times Mt$$

$mSRT$ is the input affine matrix, Ms , Mr , Mt are scale, rotation and translation output matrices.

There is something tricky about the scale transformation Ms , which can be a uniform scale transformation, or non-uniform scale transformation about an arbitrary axis. In the first case,

$$Ms = (s, s, s, 1) \times E, \text{ where } E \text{ is identity matrix, } s \text{ is a number(scaler).}$$

In the second case, things get complicated.

$$Ms = Mu \times [(Sx, Sy, Sz, 1) \times E] \times (Mu^{-1})$$

In the equation, Mu is the scale's rotational axis transform. Here is what it does on the right side of the above equation: it first rotates the space, so that scale axis is aligned with x,y,z axis, then it applies a scale transform (Sx, Sy, Sz) along the three axis, finally it rotates back to the original space by applying the inverse of Mu . For simplicity, most game engine only support uniform scale as well as scale transforms along the x,y,z axis. In other words, most game engines only deal with the situation when Mu is an identity matrix, such that Ms can be rewritten as below.

$$Ms = (Sx, Sy, Sz, 1) \times E$$

Internally, we use a 3D vector $S = (Sx, Sy, Sz)$ to represent Ms , a quaternion $R = (i, j, k, w)$ for Mr , and a 3D vector $T = (x, y, z)$ for Mt . Hence we only exported 10 degrees of freedom from the

bone matrix. Yet, the original M has 12 degrees of freedom. The missing two degrees of freedom is Mu, which we ignored and assumed them to be identity during export.

13.1.2 Exporting Mesh

So far, we have build up enough terminology to start explaining the exporting process. We will only talk about exporting animated mesh. For static mesh the process is simplified by exporting only the first frame. Terminologies which are already explained in previous section are in italic letters in the rest of the text.

There are two major passes when exporting scene data.

In the first pass, we traverse the 3ds max scene and locate the mesh object we would like to export, and then we export vertices at their *initial pose*, together with skin information, textures and materials, etc. For each vertex, we may need to collect the following information.

```
struct ModelVertex {  
    D3DXVECTOR3 pos; // 3D position  
    byte weights[4]; // bone weight of up to four bones  
    byte bones[4]; // bone index of up to four bones  
    D3DXVECTOR3 normal; // vertex normals  
    D3DXVECTOR2 texcoords; // texture coordinates  
};
```

For all meshes in the max file, we need to extract a list of textures and materials used at least once by a mesh. Also we need to extract all bones used in the scene and assign them unique IDs; parent-child relationships are also extracted for the bones. The data extracted in the first pass is mostly static data which are identical for all animation sequences of the same model.

In the second pass, we need to locate bone objects in the scene and extract animation data from each animation sequence file. This procedure is explained in detail in the next section.

13.1.3 Exporting Animation Data

Animation data is kept in bone objects. It includes snapshots of each *bone's world transformation matrix* at all frames (times) and for all animation sequences. For each bone in each animation sequence file, we need to use the following procedure to extract animation data.

```
Get the animation range of the current sequence [0, T]  
Sample the entire animation range at a given frequency, such as 30 frames per second. {  
    For each sample, extract animation as SRT keys relative to the parent bone.  
}  
Remove redundant keys
```

Of course, we can sample only when there is an explicit key in the animation range, but the above code is general enough to be applied to any animation data in 3dsmax. Next, we will show how to extract a bone's SRT keys relative to its parent bone.

Suppose we are sampling a bone at time t. First of all, we get the *world transformation matrix of the bone* at time t. We denote it as $M(t)$. Let M_i denotes the *initial bone matrix* of

the bone being sampled. M_i is a constant for the bone at all times. Now suppose there is a vertex v which is a vertex on the *mesh's initial pose* and which is fully bound to this bone. The transformed vertex position $v(t)$ at time t for v can thus be written as below.

$$v(t) = v \times M_i^{-1} \times M(t)$$

Now suppose the bone has a parent bone, whose *world transformation* and *initial bone transformation matrix* are $M_p(t)$ and M_{pi} respectively. $v(t)$ can now be rewritten as below.

$$v(t) = v \times [M_i^{-1} \times M(t)] \times \{[(M_{pi}^{-1} \times M_p(t))^{-1}] \times [M_{pi}^{-1} \times M_p(t)]\}$$

We can continue to append the identity matrix $\{[(M_{pi}^{-1} \times M_p(t))^{-1}] \times [M_{pi}^{-1} \times M_p(t)]\}$ to the end of the above equation for all parent bones until the last one in the bone hierarchy (i.e. the root bone). Thus, $v(t)$ can be rewritten as below:

$$v(t) = v \times [M_i^{-1} \times M(t)] \times \{[(M_{pi}^{-1} \times M_p(t))^{-1}] \times [M_{pi}^{-1} \times M_p(t)]\} \times \{[(M_{ppi}^{-1} \times M_{pp}(t))^{-1}] \times [M_{ppi}^{-1} \times M_{pp}(t)]\} \dots \{[(M_{root_i}^{-1} \times M_{root}(t))^{-1}] \times [M_{root_i}^{-1} \times M_{root}(t)]\}$$

We regroup the terms on the right side and rewrite $v(t)$ in the following final format.

$$v(t) = v \times \{[M_i^{-1} \times M(t)] \times [(M_{pi}^{-1} \times M_p(t))^{-1}]\} \times \{[M_{pi}^{-1} \times M_p(t)] \times [(M_{ppi}^{-1} \times M_{pp}(t))^{-1}]\} \times \{[M_{ppi}^{-1} \times M_{pp}(t)] \times [(M_{pppi}^{-1} \times M_{ppp}(t))^{-1}]\} \dots [M_{root_i}^{-1} \times M_{root}(t)]$$

Observe that the first term in the curly brackets in the above equation is:

$$rM(t) = [M_i^{-1} \times M(t)] \times [(M_{pi}^{-1} \times M_p(t))^{-1}] \quad (1)$$

$rM(t)$ is actually a relative transformation matrix from the parent bone to the current bone. Hence $rM(t)$ is the matrix we need to decompose and export. So for each bone at time t , we need to get the initial bone matrix of itself M_i and its parent M_{pi} , and we also get the world transformation matrix at time t of itself $M(t)$ and its parent $M_p(t)$. These four parameters can be directly obtained from the 3DXI API. Finally, we use equation (1) to get the relative parent-to-child bone transformation matrix.

We can now directly *decompose this affine matrix* $rM(t)$ using the 3dsmax function `decompose_affine()` to get the SRT keys which are used in the game engine. However, recall that in our character animation system, each bone has a constant pivot point. If we decompose $rM(t)$ directly, we are assuming that the mesh's origin $(0,0,0)$ is the pivot point for all bones. Obviously this assumption is wrong. A bone's pivot point should be the bone origin $(0,0,0)$ after transformed to the *initial bone pose*. In other word, the pivot point $Pivot$ for a given bone should be

$$\begin{aligned} Pivot &= (0,0,0,1) \times M_i \\ P &= M_{4 \times 4}(Pivot) \end{aligned}$$

Let P denotes the homogenous 4×4 translation matrix from $Pivot$. Taking the bone's pivot point in to consideration, we can rewrite rMt as in equation (1) as below.

$$rMt = P^{-1} \times mSRT \times P$$

$$\text{or } mSRT = P \times rMt \times P^{-1} = P \times [M_i^{-1} \times M(t)] \times [(M_{pi}^{-1} \times M_p(t))^{-1}] \times P^{-1}$$

$mSRT$ is the final affine matrix to be decomposed into scale, rotation and translation keys, which are used in the game engine, i.e. $mSRT = M_s \times M_r \times M_t$.

To summarize, for each animated bone we export a constant pivot point, and for each key frame of the bone, we export a scale key $S = (Sx, Sy, Sz)$ from M_s , a quaternion key $R = (i, j, k, w)$ from M_r , and a translation key $T = (x, y, z)$ from M_t . There are totally 10 degrees of freedom exported for each key frame of a bone.

13.1.4 Importing Animation

The majority of this chapter talks about model exporting. In this add-on section, we will talk about importing animation data from motion capture devices. Generally speaking, this should be a documented process which involves the following activities.

- Planning: write down the specifications of each character, such as name, bone hierarchy, frame rate, and make a list of actions to be captured for each character, including whether it is looped, animation length, etc.
- Shooting: we can shoot real person using motion capture devices, or virtual characters using the game engine. The second one enables the game engine's built-in character editor to export motion capture files to other editing environment such as Character Studio in 3dsmax. This can be very useful for sharing animations across different authoring platforms and different characters.
- Processing: we convert motion capture data to skeletal animation data. During this processing, one may need to correct unfit motions and make the animation ready to be used by the game engine.
- Exporting animation to the game engine: we assign skeletal animations to our character and export them to model files used by the game engine.

13.1.4.1 Motion Capture File Format

Table 13.1 outlines several motion capture formats in use today along with URLs for additional formatting information. It should be pointed out that, so far, motion capture formats are not quite standardized. But the good news is that most of the widely used motion capture file formats are in straight forward text encoding and is designed to be human readable.

Table 13.1 Motion Capture File Format

File Ext	Company	Key Style	File Format Reference
BVH & BVA	BioVision	Rotation	BVH.rtf in 3dsmax CD
CSM	3dsmax	Position	CSM.rtf in 3dsmax CD
ASF & AMC	Acclaim	Rotation	http://www.darwin3d.com/gamedev/acclaim.zip

There are two key styles which are joint rotations and marker positions. They denote how animation keys are stored in the file. Storing animation keys as point (marker) positions is

more close to the raw data obtained from motion capture devices, because motion capture devices record live motions by tracking a number of marker points (usually at the joint or pivot) over time, which are translated into a 3 dimensional digital representation. In order to capture more than one degree of freedom of the joint motions, there are usually more marker points than bones. The process of deriving joint rotations from marker position tracks is a complicated process. Hence we suggest the game engine work with file formats that store motions as joint rotations. This leaves the process of translating point positions to rotational data of bones, to motion device vendors and/or dedicated software tools like character studio in 3dsmax.

13.1.4.2 BVH File Format

We will look at the BVH format in more details. BVH is the only natively supported file format by character studio's biped system in 3dsmax. Biped provides direct input of BVH files from disk, including comprehensive key-frame reduction and footstep extraction to provide a fast and accurate means of import for large volumes of rotational data stored in the BVH format. During import, XYZ Euler joint rotations stored in the BVH file are used to derive Quaternion bone rotation data for the biped at each frame. Once imported, BVH-based animations can be saved as native Biped .bip files, providing access to a comprehensive set of animation, layered editing, retargeting and structural modification features that are built directly into Biped.

As we mentioned, motion capture formats are not quite standardized. In order for 3dsmax Biped system to understand it, all bones in the BVH format must stick to a fixed naming convention as well as bone hierarchy (i.e. parent-child relationships of named bones). The hierarchy is given below.

```
Hips {
    LeftHip (LeftUpLeg) {
        LeftKnee (LeftLowLeg) {
            LeftAnkle (LeftFoot) {
                End Site {}
            } } }
    RightHip (RightUpLeg) {
        RightKnee (RightLowLeg) {
            RightAnkle (RightFoot) {
                End Site {}
            } } }
Chest {
    LeftCollar {
        LeftShoulder (LeftUpArm) {
            LeftElbow (LeftLowArm) {
                LeftWrist (LeftHand) {
                    End Site {}
                } } }
    RightCollar {
        RightShoulder (RightUpArm) {
            RightElbow (RightLowArm) {
                RightWrist (RightHand) {
                    End Site {}
                } } }
}
Neck {
```

```

Head {
    End Site {}
}
}

```

Since 3dsmax 8, several more bone names, such as Toe, ChestN and FingerN, etc are also recognized. For more information, please refer to the BVH documentation in 3dsmax CD.

BVH allow us to specify both translation and rotational animation data; however, 3dsmax only uses the translation data in the hip bone, translations in other bones are ignored. So for the hip bone, we export vector3 translation key and a vector3 rotation key; for the rest of the bones we only export the rotation key.

The BVH serialization code example is given in the code section.

13.1.4.3 Why supporting Motion Capture Format

Motion Capture File provides a simple mean for sharing animations in different animation processing systems. In ParaEngine, we allow any in-game skeletal animations to be exported as BVH file. By editing a marker file, we can extract only specific biped bones and export common BVH files which can be further processed by 3dsmax character studio or other systems that recognize them. The work flow is illustrated in Figure 13.4.

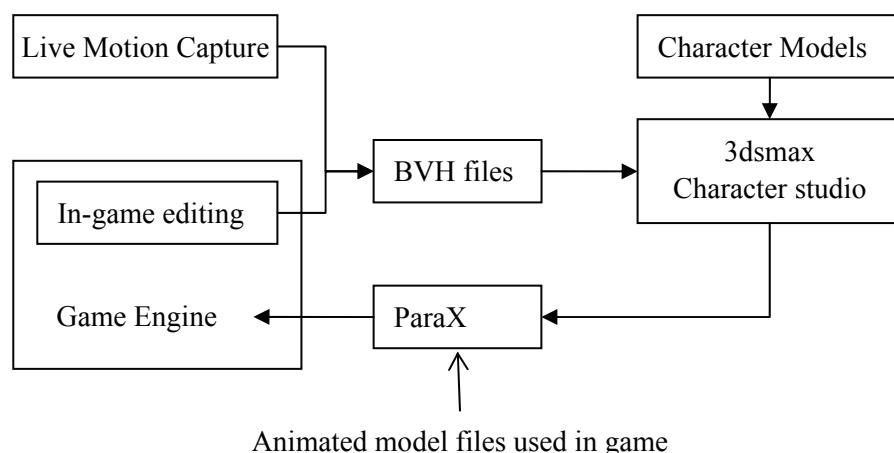


Figure 13.4 Animation Work Flow

13.2 Architecture

In this section, the software architecture of a model exporter is given. The main exporting code is organized in three places, CXFileExporter, CMaxPipeline and CParaXBuilder. They are for exporter interface, scene data extraction in 3dsmax, and serialization of model data, respectively. They are explained one by one next.

13.2.1 Exporter Interface

CXFileExporter is an implementation of 3dsmax exporter interface. It is the entry point when the user specifies the exporting options and initiates the exporting progress based on user options.

Figure 13.5 shows the common options supported by model exporter. For example, it allows a user to specify the output format, whether to export animations, and whether to compress texture files, etc. Moreover, it supports an intelligent export option, which allows users to export all 3dsmax model files in a given directory and subdirectories without any user intervention according to some predefined rules. This can be useful when we are dealing with a large game project where an automated resource build process is very important.

For each 3dsmax model file, the CXFileExporter class will generate an option object to be passed down to the export pipeline. The pipeline will extract, optimize, convert, and then the serialize data according to the given option.

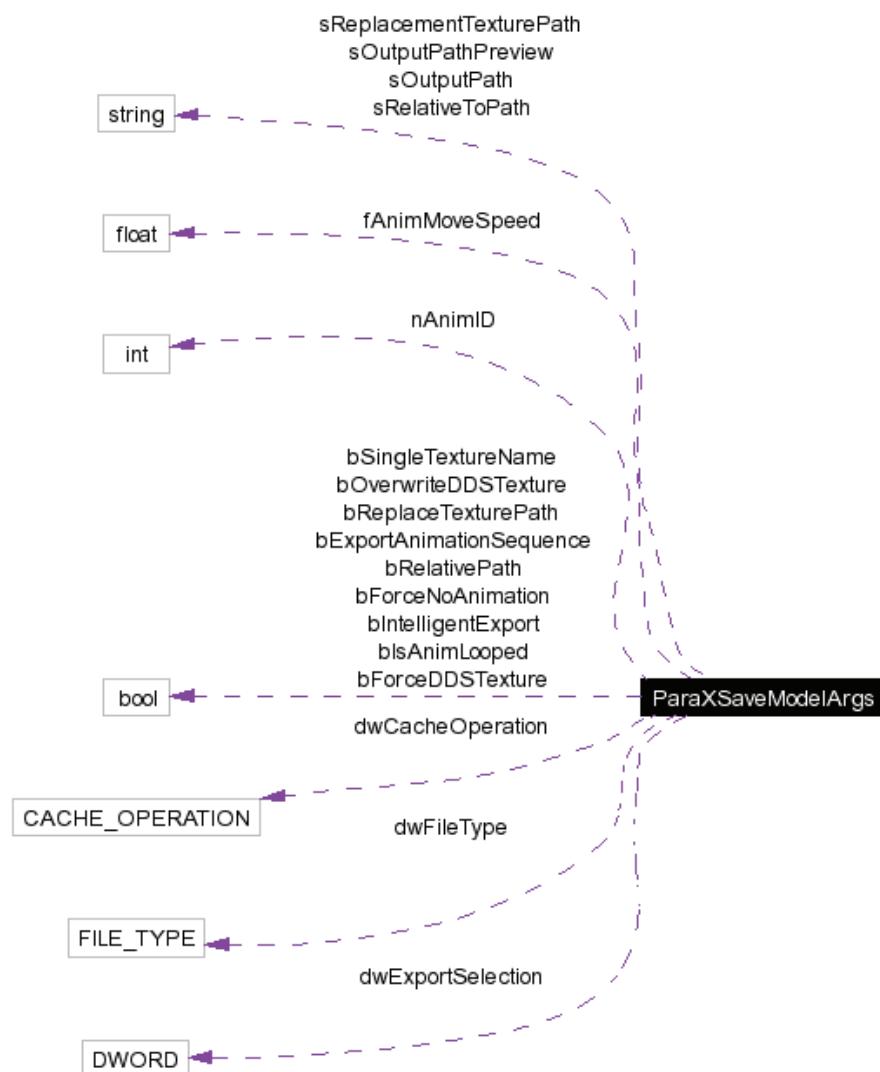


Figure 13.5 Model Export Options

13.2.2 Scene Data Extraction

CMaxPipeline is called to generate all the intermediary data through the 3dsmax's export API interface or 3DXI, which may include materials, meshes and frames (bones), etc. One can refer to a sample in DirectX 9 SDK for extracting basic mesh data in 3dsmax. Scene data extraction is a process of recursively traversing various scene nodes in 3dsmax. During scene traversing, if a bone frame or a mesh object is found, one can use the procedure described in the Foundation section to extract animation data. Some code snippet is also given in the code section of this chapter.

13.2.3 Serialization of Model Data

In the last step of the exporting progress, we pass the intermediary data extracted from the 3dsmax pipeline to CParaXBuilder, which is our model file serialization class. It performs a series of optimizations on the data, such as texture batch sort, redundant face and vertex removal, render pass sort, vertex cache optimizations, etc. Finally, it calls one of its saving functions to save the data in a specified encoding and file type, such as DirectX text file or ParaX binary or text file. Figure 13.6 shows the memory data presentation used at serialization time.

Please also note that sometimes we need to extract information from multiple files (e.g. multiple animation sequences stored in separate files) and feed them to the same CParaXBuilder object for optimization and serialization.

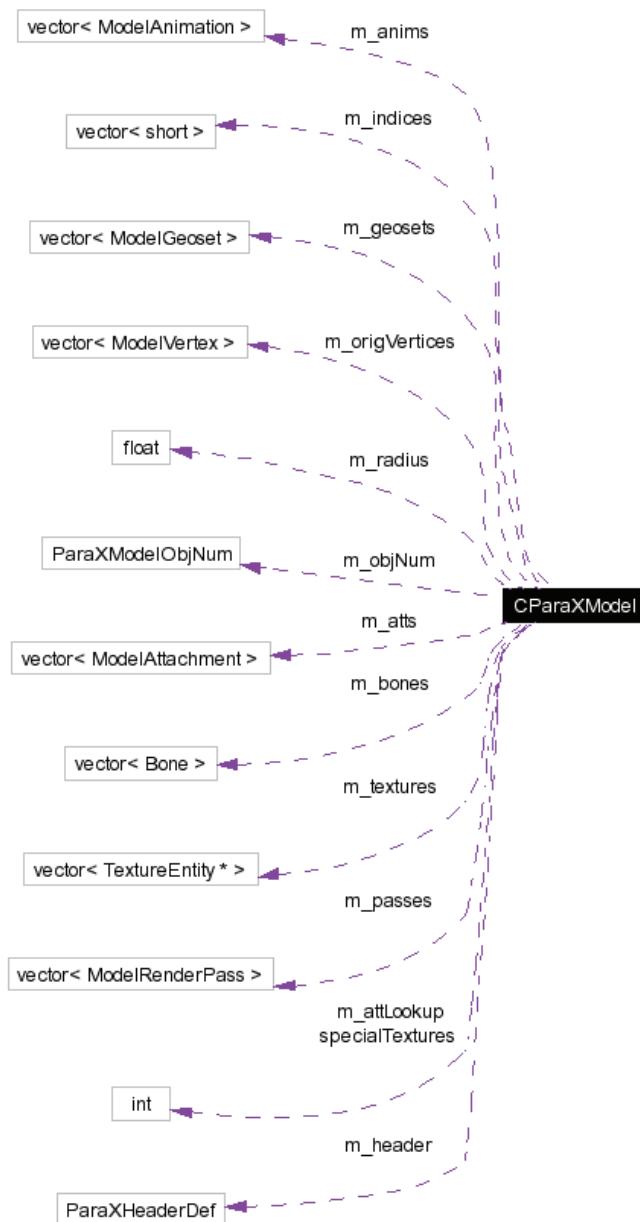


Figure 13.6 Model Data Presentation

13.3 Code and Performance Discussion

Some code snippets are given.

13.3.1 Sampling Bone Animation

The following sample code shows the sampling bone animation procedure as described in section 13.1.3.

```

void CMaxPipeline::SaveSampledTransformKeyAsSRTKey(ComPtr<IDXCCAnimationStream>& anim,
IGameNode* node, TimeValue tick, int k)
{
}

```

```

D3DXMATRIX tm;
D3DXMATRIX fm;
D3DXVECTOR3 pivot(0,0,0);
ConvertGMatrix(fm, node->GetWorldTM(tick));

if(m_curlGameSkin !=NULL)
{
    /////////////////////////////////
    // pre-multiply the inverse of the initial transform of the bone when skin is added.
    D3DXMATRIX InitBoneMatrixInverse;
    D3DXMatrixIdentity(&InitBoneMatrixInverse);

    D3DXMATRIX mInit;
    if(m_curlGameSkin->GetInitBoneTM(node, (GMatrix&)mInit))
    {
        D3DXVec3TransformCoord(&pivot, &D3DXVECTOR3(0,0,0), &mInit);
        if(k==0 && m_curFrameTracker!=0){
            m_curFrameTracker->m_pivot = pivot;
        }
        if(D3DXMatrixInverse(&InitBoneMatrixInverse, NULL, &mInit) != NULL)
            fm = InitBoneMatrixInverse*fm;
    }
    else
    {
        // pelvis does not have an initial bone transform.
    }
}
IGameNode* pn = node->GetNodeParent();
if (pn)
{
    D3DXMATRIX pfm;
    ConvertGMatrix(pfm, pn->GetWorldTM(tick));
    if(m_curlGameSkin !=NULL)
    {
        /////////////////////////////////
        // pre-multiply the inverse of the initial transform of the bone when skin is added.
        D3DXMATRIX InitBoneMatrixInverse;
        D3DXMatrixIdentity(&InitBoneMatrixInverse);

        D3DXMATRIX mInit;
        if(m_curlGameSkin->GetInitBoneTM(pn, (GMatrix&)mInit))
        {
            if(D3DXMatrixInverse(&InitBoneMatrixInverse, NULL, &mInit) != NULL)
                pfm = InitBoneMatrixInverse*pfm;
        }
    }
    D3DXMATRIX ipfm;
    if(D3DXMatrixInverse(&ipfm, NULL, &pfm) != NULL)
        tm = fm * ipfm;
}
else
{
    tm = fm;
}

DXCCKEY_MATRIX key;
key.Time = (float)k;
key.Value = tm;

```

```

// polar decomposition into SRT using decomp_affine()
AffineParts parts;
D3DXMATRIX mSRT;

if(pivot == D3DXVECTOR3(0,0,0))
{
    mSRT = tm;
}
else
{
    D3DXMATRIX mP;
    D3DXMatrixIdentity(&mP);
    mP._41 = pivot.x;mP._42 = pivot.y;mP._43 = pivot.z;
    mSRT = mP * tm;
    mSRT._41 -= pivot.x;mSRT._42 -= pivot.y;mSRT._43 -= pivot.z;
}

GMatrix tmMax;
ConvertD3dMatrix(tmMax, mSRT);
decomp_affine(tmMax.ExtractMatrix3(),&parts);
// DecomposeMatrix(tm.ExtractMatrix3(), parts.t, parts.q, parts.k);

D3DXKEY_VECTOR3 positionKey;
positionKey.Time = (float)k;
positionKey.Value = (D3DXVECTOR3)parts.t;
_V(anim->SetTranslationKey(k, &positionKey));

D3DXKEY_QUATERNION rotationKey;
rotationKey.Time = (float)k;
rotationKey.Value = (D3DXQUATERNION)parts.q;

// 3dsmax returns quaternion using column matrix. However, row matrix is used by directx.
// One can switch to row matrix by flip the quaternion's w component.
// rotationKey.Value.w = -rotationKey.Value.w;
_V(anim->SetRotationKey(k, &rotationKey));

// please note Scale's rotation axis (quaternion parts.U) is ignored.
D3DXKEY_VECTOR3 scaleKey;
scaleKey.Time = (float)k;
scaleKey.Value = (D3DXVECTOR3)parts.k;
_V(anim->SetScaleKey(k, &scaleKey));
}

```

13.3.2 BVH Serialization Code

The following code shows writing BVH animations to files.

```

bool CBVHSerializer::WriteBVHNode(int nLevel, int nBoneIndex, CParaFile* pFile,bool
bEscapeUnknownBones)
{
    int nChildCount = m_bones[nBoneIndex].GetChildCount();
    if(bEscapeUnknownBones && m_bones[nBoneIndex].m_sMarkerName == "End")
        nChildCount = 0;
    bool bIsUnknownBone = m_bones[nBoneIndex].IsUnKnownBone() && (nChildCount != 0);

    if(!bEscapeUnknownBones || !bIsUnknownBone)
    {
        // write root or joint name
        for (int i=0;i<nLevel;++i)
            pFile->WriteFormated("\t");

```

```

if(nChildCount == 0)
{
    // end site
    pFile->WriteFormated("End Site\r\n");
}
else if(nLevel == 0)
{
    // root
    if(!m_bones[nBoneIndex].m_sMarkerName.empty())
        pFile->WriteFormated("HIERARCHY\r\nROOT %s\r\n",
m_bones[nBoneIndex].m_sMarkerName.c_str());
    else
    {
        char sID[MAX_PATH];
        sprintf(sID, "%d", nBoneIndex);
        pFile->WriteFormated("HIERARCHY\r\nROOT %s\r\n", sID);
    }
}
else
{
    // joint
    if(!m_bones[nBoneIndex].m_sMarkerName.empty())
        pFile->WriteFormated("JOINT %s\r\n", m_bones[nBoneIndex].m_sMarkerName.c_str());
    else
    {
        char sID[MAX_PATH];
        sprintf(sID, "%d", nBoneIndex);
        pFile->WriteFormated("JOINT %s\r\n", sID);
    }
}
// left curly brace
for (int i=0;i<nLevel;++i)
    pFile->WriteFormated("\t");
    pFile->WriteFormated("{\r\n");

// write offset relative to parent
for (int i=0;i<nLevel+1;++i)
    pFile->WriteFormated("\t");

D3DXVECTOR3 vOffset = m_bones[nBoneIndex].m_vOffsetToParent;
if(bEscapeUnknownBones)
{
    int nIndex = GetMarkedParentIndex(nBoneIndex);
    if(nIndex >= 0 )
        vOffset = m_bones[nBoneIndex].m_vAbsOffset - m_bones[nIndex].m_vAbsOffset;
}
pFile->WriteFormated("OFFSET \t\t%f %f %f\r\n", vOffset.z, vOffset.y, vOffset.x);

// write channels
if(nChildCount != 0)
{
    for (int i=0;i<nLevel+1;++i)
        pFile->WriteFormated("\t");
    if(nLevel==0 || m_bExportBVHPosition)
        pFile->WriteFormated("CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation
Yrotation\r\n");
    else

```

```

        pFile->WriteFormated("CHANNELS 3 Zrotation Xrotation Yrotation\r\n");
    }

    for (int i=0;i<nChildCount;++i)
    {
        // this will prevent multiple end site be exported.
        int nCount = m_bones[m_bones[nBoneIndex].m_childBones[i]].GetChildCount();
        if(nCount > 0 || i == 0)
        {
            if( !bEscapeUnknownBones ||
                (m_bones[m_bones[nBoneIndex].m_childBones[i]].m_bCriticalBone))
                WriteBVHNode(nLevel+1, m_bones[nBoneIndex].m_childBones[i],
pFile,bEscapeUnknownBones);
        }
    }

    if(!bEscapeUnknownBones || !bIsUnknownBone)
    {
        // right curly brace
        for (int i=0;i<nLevel;++i)
            pFile->WriteFormated("\t");
        pFile->WriteFormated("}\r\n");
    }

    return true;
}

bool CBVHSerializer::WriteBVHNodeAnimation(int nLevel, int nBoneIndex, CParaFile* pFile,bool bEscapeUnknownBones)
{
    int nChildCount = m_bones[nBoneIndex].GetChildCount();
    if(bEscapeUnknownBones && m_bones[nBoneIndex].m_sMarkerName == "End")
        nChildCount = 0;
    bool bIsUnknownBone = m_bones[nBoneIndex].IsUnKnownBone();

    // write animation on each channel.
    if(nChildCount != 0)
    {
        if(!bEscapeUnknownBones || !bIsUnknownBone)
        {
            D3DXVECTOR3 vPos1, vPos2, vPos(0,0,0);
            D3DXVECTOR3 vRot(0,0,0);

            // if the parent bone is its immediate upper level bone, then we only need to read data from the key
            if (m_pXMesh->bones[nBoneIndex].rot.used)
            {
                Quaternion q = m_pXMesh->bones[nBoneIndex].rot.getValue(m_pXMesh->nCurrentAnim,
m_pXMesh->currentFrame, 0,0,0.f);
                q.ToRadians(&vRot.x, &vRot.y, &vRot.z);
                vRot.x *= 180.f/D3DX_PI;
                vRot.y *= 180.f/D3DX_PI;
                vRot.z *= 180.f/D3DX_PI;
            }
            if(nLevel==0)
            {
                // since we have different default orientations, rotate the root bone to face positive X.
                if(m_bRotateY)
                    vRot.y += 90.f;
            }
        }
    }
}

```

```

    }
    if(nLevel == 0 || m_bExportBVHPosition)
    {
        D3DXVECTOR3 vAbsPos = (m_pXMesh->bones[nBoneIndex].mat*m_pXMesh-
>bones[nBoneIndex].pivot);
        vPos = vAbsPos - m_bones[nBoneIndex].m_vAbsOffset;
        // if this is the root node.
        pFile->WriteFormated("%f %f %f ", vPos.x, vPos.y, -vPos.z); // the order is affected by
m_RotateY settings.
    }
    pFile->WriteFormated("%f %f %f ", vRot.z, vRot.x, vRot.y);
}
}

// write child nodes.
for (int i=0;i<nChildCount;++i) {
    // this will prevent multiple end site be exported.
    int nCount = m_bones[m_bones[nBoneIndex].m_childBones[i]].GetChildCount();
    if(nCount > 0 || i == 0)
    {
        if(!bEscapeUnknownBones ||
m_bones[m_bones[nBoneIndex].m_childBones[i]].m_bCriticalBone)
            WriteBVHNodeAnimation(nLevel+1, m_bones[nBoneIndex].m_childBones[i],
pFile,bEscapeUnknownBones);
    }
}

if(!bEscapeUnknownBones || !bIsUnknownBone) {
    // end this frame
    if(nLevel==0)
    {
        pFile->WriteFormated("\r\n");
    }
}
return true;
}

```

13.3.3 Performance Discussion

Performance is not quite an issue in an exporter program, since it is not run at production time. We can carry out the most expensive optimization in the exporter program, while leaving only light-weighted optimizations to the game engine at run time.

However, for performance reasons, we do not advise to write exporter for animated models through the scripting interface of 3dsmax. Not only will it run slowly for sophisticated meshes, but also one will lose the chance to use several free third party mesh optimization utilities, such as NVStrip from NVIDIA and DirectX's extension library, which are all in C++. For simple static mesh, programming through the scripting interface is a choice, but optional. In ParaEngine, we have a script based exporter in 3dsmax for static meshes only; and it is very handy to use and modify. Scripting is very important for rapid development of general game exporter tools. We will see it in the next section.

13.4 Summary and Outlook

Model exporter is only a member in all plug-in tools developed for 3dsmax or other content creation tools during game development. Many game studios choose to code various other

toolsets for their content creation tools, so that designers and artists can easily preview and edit in the same familiar environment. These toolsets are usually developed in an RAD fashion. And most of them are developed via the scripting system, instead of C++ API. Below are several advantages of using the scripting system for tool development:

- They have simpler interface and easy to write (no memory management, etc).
- They are more portable. One does not need to rewrite or recompile for different versions of the content tools.
- They are also quite reusable. Modules of script file can be more easily reused than objects in C++.
- They are easy to debug and modify.

Hence scripting interface becomes a good choice when you are writing plug-ins with rich GUI and without extensive computations, such as scene exporter, terrain exporter, particle system exporter, path and trigger exporter, etc. It is also a good idea to mix script based programming with C++ programming when working on your plug-in tools. Figure 13.7 shows a script based plug-in tool developed for 3dsmax in ParaEngine.

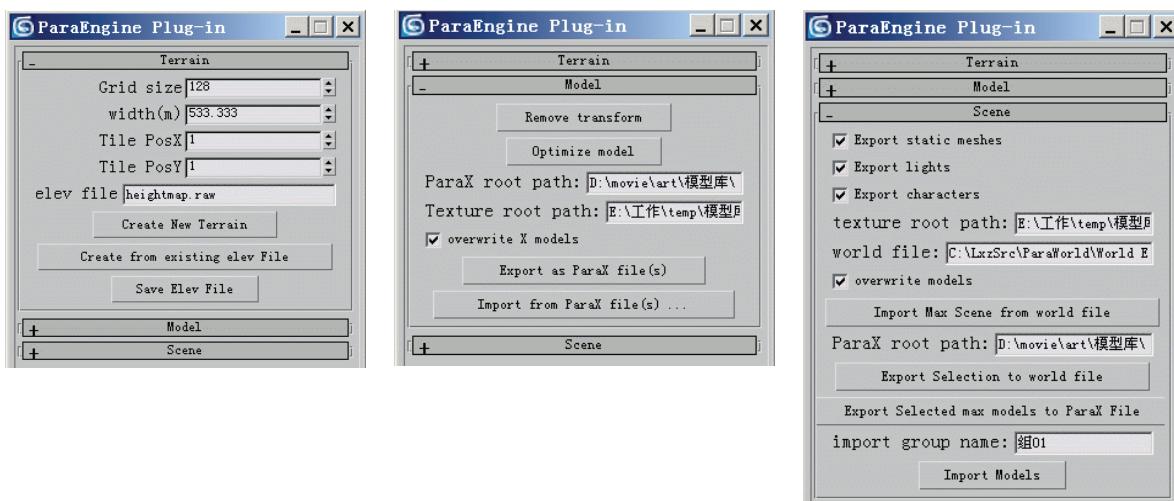


Figure 13.7 Script based ParaEngine plug-in for 3dsmax

Chapter 14 AI in Game World

Artificial Intelligence (AI) is one of the most important aspects during game development, because it is what makes a game playful. Although the topic of AI is more specific to games than to game engines, there are some paradigms which are common to many games and which need to be supported by the game engine natively.

The definition of AI has always been versatile. The scope of AI in this chapter is limited to techniques which make character behaviors in a game world customizable both before and after a game engine is released. In other words, this chapter talks about AI frameworks from a game engine's perspective. It aims to enable game designers and players to design and make intelligent behaviors of game characters.

In the Simulation chapter 8, we have already seen how game physics can affect a character's motion, such as motions that adapt to obstacles and the slope of terrain and walls, and rag-doll animations, etc. In the previous Character Animation chapter 12, we have seen how some variant finite state machine can be used to control a character's low level behaviors, i.e. motion blending of animation sequences. We will not cover them again in this chapter. Instead, we will illustrate an AI diagram that summarizes all aspects of game AI discussed in the book.

In this chapter, we will look at some common design and implementation of game AI, which makes AI in the game world customizable. These include character simulation and sensor event system, and a number of useful high level AI controllers.

14.1 Foundation

To make this chapter goal-driven, we will begin by first defining several types of character AI behaviors which is common in computer games; and then we will propose techniques and software paradigms to achieve these goals with the minimum computation.

14.1.1 Reactive Character

Reactive character is only active in the game world when some one else, usually another player, triggers it, such as mouse-clicking on its figure. In other domain of computer science, we usually call such a thing reactive agent. If we look around in other software systems, we will see that many of them are built in the reactive agent paradigm, such as web services, remote procedure calls, etc.

There are two things to consider when implementing a reactive agent based game character. One is agent discovery, i.e. how to locate a game character in a vast game world. Another is communication protocols, i.e. how one can send an eligible message to a game character, so that it can reply accordingly.

Reactive NPC is the most commonly used and the least computational expensive character in computer games. We can construct many interesting game scenarios using it, such as a merchant character standing behind the bar that sells goods to visitors, or a guard that stands at the gate of a village giving advices or quests to passers-by, etc.

14.1.2 Movie Character

Movie characters are game characters that behave according to some predefined sequence. Some cinematic characters that tell players a story between game scenes are of such a category. We can also introduce some randomness and looping mechanism in the sequence; and with this technique, we can make citizens in a game city quite convincing without simulating them using more computational expensive techniques.

Movie characters can be endowed with reactive character abilities to form more interactive and intelligent hybrid character types. For example, when you click on a patrolling character in a typical RPG game, it will stop and reply to you, and then continue with its patrolling sequence.

14.1.3 Perceptive Character

Perceptive character is a game character that can perceive other game entities when they are in its perceptive radius. By perceiving the environment, a perceptive character usually acts autonomously. Re-spawning creatures in games are usually of this type. For example, when a creature sees a player, it may choose to attack, defense, or flee, etc. Perceptive characters are usually associated with internal rules and can actively issue actions according to their perceptions.

One inevitable computation complexity of all perceptive characters is to effectively find all other characters in its perceptive radius. To reduce computation, we usually assign characters to different groups and define which groups can be perceived by a given character. Using group policy, we can easily reduce large number of irrelevant characters which are in a character's vicinity, yet of no interest to it. We will see them in details in the next section.

14.1.4 Character Simulation and Sensor Events

In previous sections, we have identified several types of game characters. A game engine must be able to simulate characters according to their AI requirements. In ParaEngine, we use a unified character simulation and sensor event framework which could achieve any combination of the aforementioned game character types. To begin with, we will define a general game character as a group of attributes, functions and sensor events. For ease of programmer readers, we will describe it in a class called `IGameObject`, from which other specific types of characters may be derived. See below. The sensor events are in bold text. They will be automatically called during game world simulation. One can refer to Chapter 8.2.2 for the character simulation code.

```
class IGameObject
{
public:
    /** call back type */
    enum CallBackType{
        Type_EnterSentientArea=0,
        Type_LeaveSentientArea,
        Type_Click,
        Type_Event,
        Type_Perception,
        Type_FrameMove,
        Type_OnLoadScript
    };
}
```

```

/** save the current character to database according to the persistent property.
if the object is persistent, the action is to update or insert the character to db
if the object is non-persistent, the action is to delete it from the database. */
virtual bool SaveToDB();

/** automatically generate way points according to its perceptions */
virtual void PathFinding(double dTimeDelta);

/** animate biped according to its current way point lists and speed.
* assuming that no obstacles are in the way it moves.*/
virtual void AnimateBiped( double dTimeDelta, bool bSharpTurning = false /*reserved*/);

/** Get the AI module that is dynamically associated with this object */
virtual CAIBase* GetAIModule();

///////////////////////////////
// the following must be implemented. The IGameObject will automatically activate
// the script. Hence, object which implements these functions need to call the base class
// to ensure that the script is also activated.
///////////////////////////////

/** called when this object is attached to the scene. */
virtual int On_Attached();
/** called when this object is detached from the scene */
virtual int On_Detached();

/** when other game objects of a different type entered the sentient area of this object.
This function will be automatically called by the environment simulator. */
virtual int On_EnterSentientArea();
/** when no other game objects of different type is in the sentient area of this object.
This function will be automatically called by the environment simulator. */
virtual int On_LeaveSentientArea();

/** when the player clicked on this object.
This function will be automatically called by the environment simulator. */
virtual int On_Click(DWORD nMouseKey, DWORD dwParam1, DWORD dwParam2);

/** TODO: Some game defined events, such as user attack, etc. */
virtual int On_Event(DWORD nEventType, DWORD dwParam1, DWORD dwParam2);

/** when other game objects of a different type entered the perceptive area of this object.
This function will be automatically called by the environment simulator. */
virtual int On_Perception();
/** called every frame move when this character is sentient.
* This is most likely used by active AI controllers, such as movie controller. */
virtual int On_FrameMove();

/** add a new call back handler. it will override the previous one if any.
bool AddScriptCallback(CallBackType func_type, const string& script_func);
/** return NULL if there is no associated script. */
ScriptCallback* GetScriptCallback(CallBackType func_type);
/** remove a call back handler*/
bool RemoveScriptCallback(CallBackType func_type);

/** whether the biped is sentient or not*/
bool IsSentient();
/** set the object to sentient.
* @param bSentient: true to make sentient. if the object's sentient count is larger than 0, this

```

```

* function has no effect; false, to remove the object from the sentient list.
*/
void MakeSentient(bool bSentient=true);

/** get the sentient radius. usually this is much larger than the perceptive radius.*/
float GetSentientRadius();
/** set the sentient radius. usually this is much larger than the perceptive radius.*/
void SetSentientRadius(float fR);

/** get the perceptive radius.*/
float GetPerceptiveRadius();
/** Set the perceptive radius.*/
void SetPerceptiveRadius(float fNewRaduis);

/** return the total number of perceived objects.*/
int GetNumOfPerceivedObject();

/** get the perceived object by index. This function may return NULL.*/
IGameObject* GetPerceivedObject(int nIndex);

/** get the distance square between this object and another game object*/
float GetDistanceSq2D(IGameObject* pObj);

/** whether the object is always sentient. The current player is always sentient */
bool IsAlwaysSentient();
/** set whether sentient.*/
void SetAlwaysSentient(bool bAlways);

/** update the tile container according to the current position of the game object.
* This function is automatically called when a global object is attached.*/
void UpdateTileContainer();

/** is global object*/
virtual bool IsGlobal(){return m_bIsGlobal;}

/** make the biped global if it is not and vice versa.*/
virtual void MakeGlobal(bool bGlobal);

/** set the group ID to which this object belongs to. In order to be detected by other game object.
* Object needs to be in group 0 to 31. default value is 0*/
void SetGroupID(int nGroup);
/** Get the group ID to which this object belongs to. In order to be detected by other game object.*/
int GetGroupID();

/** set the sentient field. A bit field of sentient object. from lower bit to higher bits, it matches to the
0-31 groups. @see SetGroupID()
* if this is 0x0000, it will detect no objects. If this is 0xffff, it will detects all objects in any of the 32
groups.
* if this is 0x0001, it will only detect group 0.
* @param dwFieldOrGroup: this is either treated as field or group,depending on the blsGroup
parameter.
* @param blsGroup: if this is true, dwFieldOrGroup is treated as a group number of which object
will detect.
* if this is false, dwFieldOrGroup is treated as a bitwise field of which object will be detected.
*/
void SetSentientField(DWORD dwFieldOrGroup, bool blsGroup=false);
/** @see SetSentientField*/
DWORD GetSentientField();

```

```

    /** return true if the current object is sentient to the specified object. If the object is always sentient,
this function will always return true.*/
    bool IsSentientWith(const IGameObject * pObj);

    /** whether the object is persistent in the world. If an object is persistent, it will be saved to the
world's database.
if it is not persistent it will not be saved when the world closes. Player, OPC, some temporary
movie actors may be non-persistent; whereas NPC are usually persistent to the world that it belongs
to.*/
    bool IsPersistent(){return m_bIsPersistent;};
    /** whenever a persistent object is made non-persistent, the SaveToDB() function will actually
remove it from the database and the action can not be recovered.
* so special caution must be given when using this function to prevent accidentally losing
information.
@see IsPersistent()
void SetPersistent(bool bPersistent);

    /** whether some of the fields are modified. It is up to the implementation class to provide this
functionality if necessary.*/
    virtual bool IsModified();
    /** set whether any field has been modified.*/
    virtual void SetModified(bool bModified){m_bModified = bModified;};
}

```

The above codes are explained in the following sections as character attributes and events.

14.1.4.1 Character Simulation Attributes

The most important attributes of a common game character are “global”, “Sentient”, “AlwaysSentient”, “Sentient Radius”, “Perceptive Radius”, “Group ID” and “SentientField”. We will explain them one by one next.

“global” is a Boolean specifying whether the character can move in the scene. The “global” attribute is usually assigned manually when the character is created. The computation to simulate a non-global or static character is far more economic than to simulate a global one, which can move in the scene autonomously. Pure reactive character is a typical non-global character.

“sentient” is a Boolean which is automatically set on and off during character simulation. It specifies whether a game character needs to be simulated in the current frame. A huge game world may be populated with tens of thousands of characters, but most of them are not sentient. Consider a city with 1000 NPC and creatures. If the human player’s avatar is miles away from them, why bother to simulate them at all? On each simulation step, the character simulator can automatically wake up non-sentient characters and put sentient ones to sleep according to the values in the “sentient field” and “group ID” attributes of characters, as well as their relative positions. The character simulation code in Chapter 8.2.2 shows that the simulation time at a given frame is only linear to the number of sentient characters currently in the scene. It has nothing or little to do with the total number of loaded game characters. In other words, the character simulation complexity is $O(N)$, where N is the number of sentient game objects currently in the scene.

“AlwaysSentient” is a Boolean telling whether a certain character should always be sentient. The simulator will never make an “AlwaysSentient” character non-sentient. Always sentient characters in a game are usually the origin of the AI logic triggering chains. The most common “AlwaysSentient” character is the human player’s avatar in the game world. When

an “AlwaysSentient” character is in the scene, the character simulator will wake up all other relevant characters which can sense it and in its vicinity. In a RTS (real-time strategy) game, all game units may need to be always sentient; in a RPG game, only the main player need to be always sentient; and in a MMORPG (massively multiplayer online role playing game), the game server may need to treat all clients’ avatars and the Boss NPCs as “AlwaysSentient” characters and everything else, such as ordinary NPC, re-spawning creatures as not “AlwaysSentient”.

“Sentient Radius” is an important radius that the character simulator used to generate sensor events for a character. The sentient radius is usually a large radius, such as 50-100 meters or it can be proportional to the camera view frustum radius. The spherical region specified by a character’s sentient radius is used to wake up other non-sentient objects in this range. Consider an “AlwaysSentient” player walks along a path, it will keep waking up all other non-sentient objects whose sentient area intersect with it; whereas the character simulator also keeps putting sentient objects to sleep when their sentient area are no longer inside that of any other interested sentient characters. See Figure 14.1. Two events are fired by the character simulator whenever a character changes states between sentient and non-sentient. They are `On_EnterSentientArea` and `On_LeaveSentientArea`. The `On_FrameMove` event and various other AI modules (which we will see in later sections) are called every few frames during the time when a character is sentient.

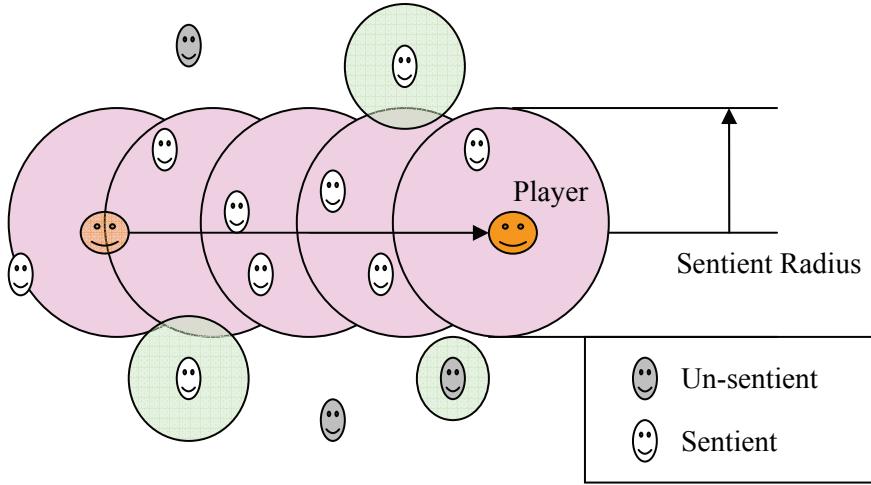


Figure 14.1 Sentient radius and swept region

“Perceptive Radius” is an important radius that the character simulator uses to generate sensor events for a character. The character simulator will call `On_Perceive` event of a given character on a simulation frame if and only if (1) the character is sentient (2) there are other interested characters in the perceptive radius of this character. Hence, the perceptive radius is always smaller than the sentient radius. The reason that we need both sentient and perceptive radii is that we can have two sets (layers) of AI functions at different ranges. One can define more radii if one likes. But two layers of AI at different ranges are generally enough for most game scenarios. For example, in the first layer or within the sentient radius, the character just moves using some predefined patterns, such as patrolling. In the second layer or within the smaller perceptive radius, the character may issue actions according to its perceptions, such as attacking a player, etc.

“Group ID” is an integer from 0 to 31, denoting 32 groups. Any character in the scene is assigned to one of the 32 groups. By assigning group ID to characters, we can define relationships between characters from different and/or the same groups. For example, we can define rules such that characters from group 0 can only perceive characters from group 1 and group 2. Such group relationship information is encoded in a 32bit DWORD attribute called “SentientField” (see the next paragraph). In an actual game, we may put all players to one group, all network players to one group, all static task NPCs to one group, all ally creatures to one group, all enemy creatures to one group, all cinematic movie character to one group, all dummy NPC to one group, etc. The reason to partition game characters by groups is that the character simulator only needs to care about pair of characters whose group IDs are related. This will greatly reduce the total number of sentient characters in the scene as well as number of interested (relevant) characters in the vicinity of any sentient character.

“SentientField” is a 32bits DWORD, from lower bits to higher bits, each denoting one of the 32 groups. If any bit of the “SentientField” is 1, it means that the character is able to sense characters belonging to that group; and if any bit is 0, it means that the character can not sense or is irrelevant to characters from the corresponding group. For example, with sentient field 0x1, the object is only interested in or sentient with group 0; with sentient field 0xffffffff, the object is interested in characters of any group. Table 14.1 shows several typical game character types and their typical group ID and sentient field values.

Table 14.1 Typical game characters and their group ID and sentient field

S-field\G-ID	<i>Player</i>	<i>Movie Character</i>	<i>Static NPC</i>	<i>Enemy creatures</i>
	Group 0	Group 1	Group 2	Group 3
<i>Bit 0</i>	1	1	0	1
<i>Bit 1</i>	1	0	0	1
<i>Bit 2</i>	1	0	0	1 or 0
<i>Bit 3</i>	1	0	0	0
Sentient Field	0xF	0x1	0x0	0x7 or 0x3

14.1.4.2 Character Simulation Events

There are five typical events that the character simulator calls automatically. The character AI code can be written directly in them or be called from them.

Table 14.2 Character simulation events

On_Load	Called when the character is loaded from disk or database to the scene. Although it is not called by the character simulator, it looks and functions like a simulation event.
On_EnterSentientArea	When other game objects of a different type entered the sentient area of this object. This function will be automatically called by the character simulator.

On_LeaveSentientArea	When no other game objects of different type is in the sentient area of this object. This function will be automatically called by the character simulator.
On_Click	When the player clicked on this object. This function will be automatically called by the character simulator.
On_Perception	When other game objects of a different type entered the perceptive area of this object. This function will be automatically called by the character simulator.
On_FrameMove	This function is called every frame when the character is sentient. This is mostly used by AI controllers.

Each event can be optionally associated with a script file. Whenever an event is fired, the associated script is also called. We have not talked about scripting yet, but one can think of them as programs to be compiled and executed at runtime. One can have a quick glance at the scripting chapter for a brief introduction to game scripting technology. In this chapter, we will only look at C++ or hard-coded AI code in the game engine. In the following sections, we will see using AI controllers in the On_FrameMove events to achieve a variety of character behaviors.

14.1.5 AI Controllers

AI controller is a reusable C++ AI code pattern. We will examine four kinds of AI controllers in this chapter, they are facing tracking controller, follow controller, movie controller and sequence controller. A character may be assigned any number of AI controller instances at runtime. There are some internally defined priorities between different kinds of AI controllers. The character simulator, which we have examined in previous section, automatically executes all the AI controllers that are associated with a sentient character at each simulation frame.

All AI controllers have a virtual function called FrameMove(), which are called automatically per frame. We can think of it being called inside or just before On_FrameMove() event of the character. Because the FrameMove() function call of AI controllers originate from the character simulator, the code inside FrameMove() can access all perceived characters in its vicinity. Some controllers that we used in ParaEngine are given next.

14.1.5.1 Face Tracking Controller

Face tracking controller controls a character to always face a given target or another character. Face tracking controller has relatively high priority. The implementation is very simple. In the FrameMove() function of the controller, it just retrieve the closest character from its perceived character list and set the facing accordingly. This controller is commonly used when a player approaches an NPC. The NPC with a face tracking controller will automatically rotates its neck to face the incoming player.

14.1.5.2 Follow Controller

Follow controller controls a character to follow another character as long as the target is in sight. The implementation is straight forward. It first searches if the target is in sight by examining in its perceived character list. If the target is perceived, it just commands the character to walk towards it until some minimum distance to the target is reached.

14.1.5.3 Movie Controller

Movie controller controls a character to act according to some predefined action script. The script is a list of time and action pairs. Movie controller is suitable for playing back time accurate character motions. The implementation is very similar to the key framed character animation system, except that the keys are the character positions in the scene, the animation IDs to play, the dialogs to speak, etc.

14.1.5.4 Sequence Controller

Sequence controller controls a character according to some predefined sequenced commands list. It executes at most one command per frame, and it will only execute the next command when the current command is finished or timed out. Sequence controller is similar to movie controller, except that its progression is dependent on the termination of a command, instead of time. One can think of a sequence controller as a short computer program. The program may contain branch and loop operations. Instead of using a separate C++ thread to execute this program, the sequence controller executes at most one command per simulation step in the main thread. The following code shows a list of common instructions supported by a sequence controller.

```
/** run to a position relative to the current position.*/
void RunTo(float x,float y,float z);
/** walk to a position relative to the current position.*/
void WalkTo(float x,float y,float z);
/** move (using the current style i.e. walk or run) to a position relative to the current position.*/
void MoveTo(float x,float y,float z);
/** play an animation by animation name*/
void PlayAnim(const string& sAnim);
/** play an animation by animation id*/
void PlayAnim(int nAnimID);
/** wait the specified seconds, without further processing commands*/
void Wait(float fSeconds);
/** execute a given script command*/
void Exec(const string& sCmd);
/** pause the sequence infinitely until some one resumes it.*/
void Pause();
/** turn the character to face a given absolute direction in radian value.*/
void Turn(float fAngleAbsolute);
/** move forward using the current facing a given distance*/
void MoveForward(float fDistance);
/** move backward using the current facing a given distance*/
void MoveBack(float fDistance);
/** move left using the current facing a given distance*/
void MoveLeft(float fDistance);
/** move right using the current facing a given distance*/
void MoveRight(float fDistance);
/** Jump once*/
void Jump();
```

```

/** offset the current sequence commands by a given steps. */
void Goto(int nOffset);
/** offset to a label, if label not found, it will wrap to the beginning. */
void Goto(const string& sLabel);
/** add a new label at the current position. */
void Lable(const string& sLabel);

```

Sequence controller can be combined with character sensor events to build fairly complex AI logics. A sequence controller alone can be used to model behaviors such as patrolling guards, wandering enemy creatures, busy citizens on a city street, etc. Sequence and movie controllers are usually used via the scripting interface. Although we have not covered scripting in game engine, one can still easily preview the script code example of a sequence controller. See below.

```

local s = player:ToCharacter():GetSeqController();
s:BeginAddKeys();
  s:Lable("start");
  s:PlayAnim("EmotePoint");
  s:WalkTo(10,0,0);
  s:Wait(0);
  s:Turn(-1.57);
  s:WalkTo(-5,0,0);s:Jump();s:RunTo(-5,0,0);
  s:Exec(";NPC_SEQ_EXEC_Test()");
  s:Goto("start");
s:EndAddKeys();

```

14.1.6 Frame Rate of AI Logics

The character simulation engine is executed at a high frame rate, such as 30 FPS. However, not all AI logics need to be executed at such a high rate, we can save three to five times computation by simply adjusting the frame rate of some AI modules to a lower value, such as 10FPS or 6 FPS, or even lower depending on the type of AI concerned. On the other hand, distributing AI code into several frames is also a very good idea.

However, if the same character is subject to several AI modules each of which are operating at different frame rates, there could be some tricky issues to deal with. Figure 14.2 shows the different game engine modules which may affect the behavior of a simulated character. AI module with high priority will override behaviors generated by AI modules with low priority. AI modules with high priority usually executes at a slightly higher frame rate than those with low priority.

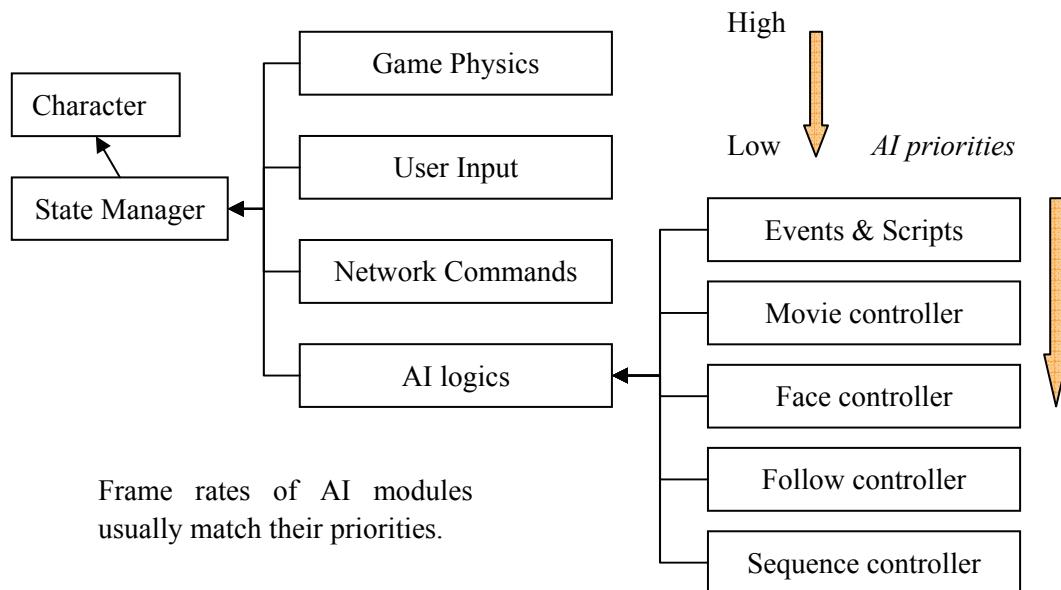


Figure 14.2 Frame rate and priority of AI modules

Implementing a low priority AI module could get increasingly complicated when there are more high priority AI modules working together. Take the sequence controller for example. It has very low priority and frame rate. Many other controllers may override the behavior generated by a sequence controller. For example, a sequence controller controls a character walking in circles; then suddenly the character perceives a player, the follow controller is activated and the character begins to walk after the player instead of doing its routine circling; after a while, the character follows the player into a deep hole on the ground; the game physics engine immediately detects this and it works with the character state manager to play a falling down animation; after falling down the hole, the character loses sight of the player, and it tries to go back to its initial position and continue with its walk in circles there. In this long process, the sequence controller is suppressed by a number of high priority modules and for quite a long time.

Another challenge of implementing low frame rate AI is that we need to make more predictions to the motions of the character in order to generate smooth animation under the normal render frame rate. Consider the following consecutive commands in the sequence controller.

```
s:WalkTo(10,0,0);
s:WalkTo(0,0,10);
```

What we intended is that the character should walk 10 units along the x axis, and then 10 units along the y axis. This works fine if the sequence controller is running at the same frame rate as the character simulator. However, if the sequence controller is only executed every 0.2 seconds or at 5 FPS, the result can be very different from what is expected. Without proper prediction, the character will first walk 10 units along the x axis, and before the next walk command is issued 0.2 seconds later, the character state manager determines that the character has already reached its destination and a standing animation will be wrongly inserted.

This problem is caused by different AI modules working at different frame rates. A workaround to this problem is to predict as much as possible. For example, in the sequence controller, we need to predict 0.2 seconds ahead. I.e. if the character is going to reach its destination in the next 0.2 seconds, we will immediately proceed to the next command without waiting till the next frame. Thus, the termination of any command in sequence controller is defined as below: a command terminates if it is going to meet its termination criterion in the next 0.2 seconds. By predicting 0.2 seconds ahead of time, the sequence controller can avoid idle frames that induce unnecessary animations. However, it also means that all commands of a sequence controller may be terminated at most 0.2 seconds earlier. If we want to carry out a command to its accurate completion, we can insert a wait(0) command (an empty command), which tells the sequence controller to wait 0 seconds (it actually waits 0.2 seconds depending on the FPS of the sequence controller). See below.

```
s:WalkTo(10,0,0);
s:Wait(0);
s:Turn(3.14);
s:WalkTo(0,0,10);
```

Of course, if the sequence controller in your implementation is a high priority AI module, the above Wait(0) command is not necessary.

14.1.7 Summary of AI

Game AI is very versatile. Due to the limited computation resources, game AI are usually designed in a layered or prioritized fashion, in which different modules may be running under different frame rates. Another characteristic of game AI is that the AI codes are usually not centralized in a single all-in-one module. Modern computer game engines features real-time physics and autonomous animations, which forces the AI (or character behavior related code) to be distributed in various places of the game engine. Script file is another place that AI code is usually placed in. This is especially true for RPG games with large number of unique game characters. With the up coming of Internet enabled games, AI logics is going to be distributed on the network.

14.2 Architecture

This section shows the basic architecture of the AI framework discussed in the chapter.

14.2.1 Game Object Base Class

All simulated characters need to be derived from a base game object or IGameObject. Figure 14.3 shows an example. The biped object is derived from IGameObject, and an RPG character is derived from a biped object. IGameObject exposes a common interface used by the environment simulator. The interface is already shown in the previous Foundation section.

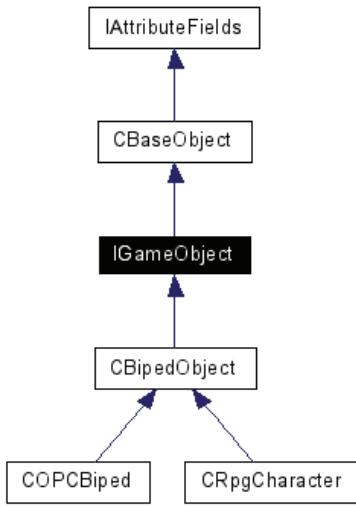


Figure 14.3 IGameObject inheritance graph

14.2.2 AI Controller

All AI controllers are also derived from a common base class called CBipedController as shown in Figure 14.4.

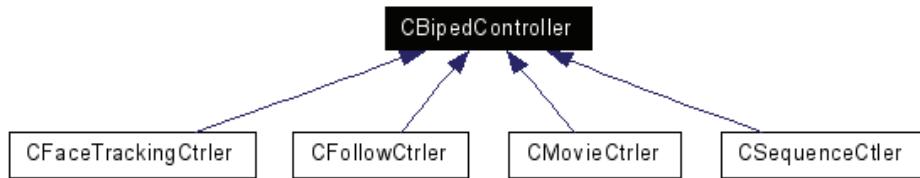


Figure 14.4 AI controller inheritance graph

The following shows the CBipedController interface. The most important function of a biped controller is the frame move function which is called by the character simulator every frame if the associated character is sentient.

```

/*
 * Base Interface for Biped controller. The AI module contains a collection of biped controllers
 * for directing the behavior of the biped. Each biped controller derived from this class may
 * control a certain aspect of the biped, such as UserController, MovieController, OPCController
 * NPLController, FacingController, CollisionController, etc.
 */
class CBipedController
{
public:
    CBipedController(CAIBase* pAI);
    CBipedController(void);
    virtual ~CBipedController(void);

private:
    /** whether the controller is suspended, so it does not take effects on the next frame move.*/
    bool m_bSuspended;
    /** to which this biped controller is associated.*/
    CAIBase* m_pAI;
}

```

```

protected:
    /** total time elapsed since the controller is active (not suspended). */
    float m_fTimeElapsed;

public:
    /** check whether the controller is active(not suspended).*/
    bool IsActive();
    /** suspend the controller, so it does not take effects on the next frame move.*/
    virtual void Suspend();
    /** resume the controller. */
    virtual void Resume();
    /** set the internal timer. This is useful when the behavior is relevant to the timer.*/
    virtual void SetTime(float fTime);
    /** get the current movie time*/
    float GetTime();
    /** get the biped in the perceived biped list, which is closet to the current biped.
     * @param plnput: the active biped list.
     * return: NULL is returned if there is no closest biped. The pointer returned is only valid where
     plnput is valid*/
    IGameObject* GetClosestBiped(IGameObject* plnput);

    /** Find a biped with a matching name. * is supported, which matches any character(s).
     * @param plnput: the active biped list.
     * @param searchString: e.g. "LiXizhi", "LXZ"
     */
    IGameObject* FindBiped(IGameObject* plnput, const std::string& searchString);

    /** get the biped, to which this biped controller is associated. */
    CBipedObject* GetBiped();
    /** set the AI object to which this object is associated.*/
    void SetAI(CAIBase* pAI);

    /** a virtual function which is called every frame to process the controller.
     * @param fDeltaTime: the time elapsed since the last frame move is called.
     */
    virtual void FrameMove(float fDeltaTime);
};

```

The collaboration of AI controllers with the character scene object and the character state manager is illustrated in the collaboration diagram shown in Figure 14.5. CAIBase and its derived classes are AI controller containers. For example, a typical NPC AI module may contain four AI controllers as shown in Figure 14.6.

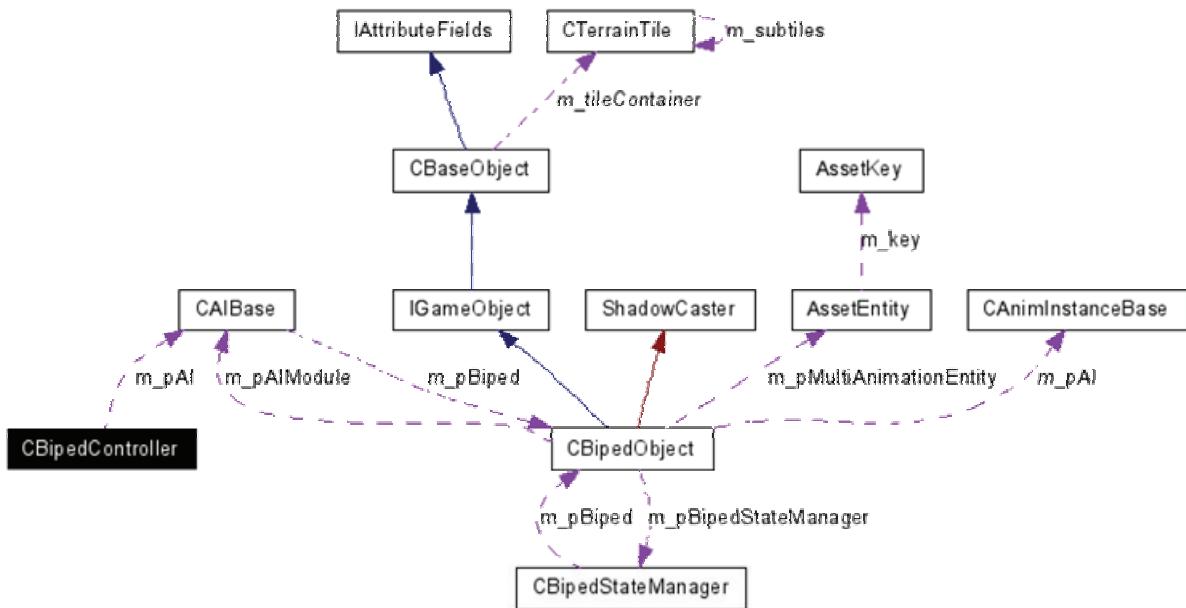


Figure 14.5 Collaboration diagram for CBipedController

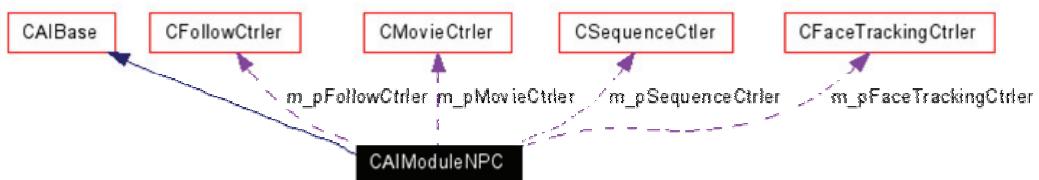


Figure 14.6 Collaboration diagram for CAIModuleNPC

14.2.3 Other AI Modules

Other AI modules in ParaEngine can be found in other chapters of this book. The following is a quick reference to those places in the book.

- Character simulator: in the simulation **chapter 8**.
 - Character state manager: in the character and animation **chapter 12**.
 - Character physics: in the simulation **chapter 8**.
 - Script based AI: in the scripting **chapter 16 and Appendix B**.
 - Path-finding: in the navigation **chapter 15**.

14.3 Code and Performance Discussion

This section provides some useful code snippets.

14.3.1 Follow Controller

Class structure

```

/** a follow controller
 * it will follow a given target or another biped.
 * please note that the controller will only follow an object within its perceptive radius. */
class CFollowCtrlr : public CBipedController
{
public:
    CFollowCtrlr(void);
    CFollowCtrlr(CAIBase* pAI);
    virtual ~CFollowCtrlr(void);

private:
    /// name of the biped to follow
    std::string m_sFollowTarget;
    /// default radius around the target biped. it will control the biped to try it best to stand on this circle.
    float m_fRadius;
    /// it will control the biped to stand beside the target with the target facing shifted by this value.
    /// note that +Pi means behind the biped.
    float m_fAngleShift;

public:
    /*
     * Follow a biped at a specified circle position.
     * @param obj: format "sName radius angle" | "sName"
     * sName: it is the name of the biped to follow,
     * radius: [optional, default to 2.5f] it is the default radius around the target biped. it will control the
     * biped to try it best to stand on this circle.
     * angle: [optional, default to Pi] it will control the biped to stand beside the target with the target
     * facing shifted by this value.
     * note that +Pi means behind the biped.
     * e.g. "lixizhi", "lixizhi 2.5 3.14", "lixizhi 3.0 0", "lixizhi 3.0 1.57", "lixizhi 3.0 -1.57"
     */
    void SetFollowTarget(const std::string& obj);
    const std::string& GetFollowTarget();

    /** a virtual function which is called every frame to process the controller.
     * @param fDeltaTime: the time elapsed since the last frame move is called.
     * @param plnput: It holds any information that is perceived by a Active Biped object
     */
    virtual void FrameMove(float fDeltaTime);
};


```

Implementation

```

void CFollowCtrlr::FrameMove(float fDeltaTime)
{
    CBipedObject * pBiped = GetBiped();
    IGameObject* pPercevied = FindBiped(pBiped, m_sFollowTarget);
    if(pPercevied!=NULL && pBiped!=NULL && pPercevied->GetDistanceSq2D(pBiped)>MIN_FOLLOW_CATCHUP_DIST_SQUARE)
    {
        float fFacing = pPercevied->GetFacing() + m_fAngleShift;
        D3DXVECTOR3 vDest;
        pPercevied->GetPosition(&vDest);

        vDest.x += cosf(fFacing)*m_fRadius;
        vDest.z += sinf(fFacing)*m_fRadius;

        CBipedStateManager* pState = pBiped->GetBipedStateManager();
        if(pState)
        {
            pState->SetPos(vDest);
        }
    }
}

```

```

    pState->AddAction(CBipedStateManager::S_WALK_POINT);
}
}
}
```

14.3.2 Sequence Controller

Class structure

```

/** A sequence controller is a biped controller which moves the biped according to some predefined
sequence.*/
class CSequenceCtler : public CBipedController
{
public:
    CSequenceCtler();
    CSequenceCtler(CAIBase* pAI);
    virtual ~CSequenceCtler(void);
public:
    /** a virtual function which is called every frame to process the controller.
     * @param fDeltaTime: the time elapsed since the last frame move is called.
     */
    virtual void FrameMove(float fDeltaTime);

    bool Save(bool bOverride);
    bool Load(int nSequenceID);
    bool Load(const string& fileName);
    int Create(const string& name, const string& description, const char* pData,bool bInMemory);
    string ToString();

    /** Get the current absolute playing cursor position*/
    int GetKeyPos(){return m_nKeyPos;};
    /** set the current absolute playing cursor position*/
    void SetKeyPos(int nPos);

    /** get total key count*/
    int GetTotalKeys();
    /** offset the key index according to the current play mode. i.e. it will automatically wrap to the
beginning if looping.
     * @param nOffset: number of keys to advance.
     * @return: the number of keys that have been successfully offseted. Usually if the returned value is
not equal to the input value, it means
     * that the sequence should be paused. */
    int AdvanceKey(int nOffset);
    /** call the command functions(RunTo, MoveTo, etc) only between the matching pair of
BeginAddKeys() and EndAddKeys()*/
    void BeginAddKeys();
    /** call the command functions(RunTo, MoveTo, etc) only between the matching pair of
BeginAddKeys() and EndAddKeys()*/
    void EndAddKeys();
    /** get sequence ID*/
    int GetSequenceID();
    /** delete keys range
     * @param nFrom: 0 based index.
     * @param nTo: 0 based index, if -1, it means the last one. */
    bool DeleteKeysRange(int nFrom, int nTo);

    /** get the play direction.*/
    bool GetPlayDirection(){return m_bForward;};
```

```

/** set the play direction. */
void SetPlayDirection(bool bForward){m_bForward = bForward;};

/** the minimum time between two successive calls. */
float GetInterval(){return m_fMinInterval;};
/** the minimum time between two successive calls. */
void SetInterval(float fInterval){m_fMinInterval = fInterval;};

/** get the starting position. */
D3DXVECTOR3 GetStartPos(){return m_vStartPos;};
/** set the starting position. */
void SetStartPos(const D3DXVECTOR3& vPos){m_vStartPos = vPos;};

/** get the start facing. usually default to 0. */
float GetStartFacing(){return m_fStartFacing;};
/** Set the start facing. usually default to 0. */
void SetStartFacing(float facing){m_fStartFacing = facing;};

/** get the current play mode */
int GetPlayMode(){return (int)m_nPlayMode;};
/** set the current play mode */
void SetPlayMode(int mode);

/** get the number of seconds after which all move commands will be treated as finished.
default value is 30 seconds. */
float GetMovingTimeout(){return m_fMovingTimeOut;};
/** set the number of seconds after which all move commands will be treated as finished.
default value is 30 seconds. */
void SetMovingTimeout(float fTimeout){m_fMovingTimeOut=fTimeout;};

///////////////////////////////
// commands:
public:
    /** run to a position relative to the current position. */
    void RunTo(float x,float y,float z);
    /** walk to a position relative to the current position. */
    void WalkTo(float x,float y,float z);
    /** move (using the current style i.e. walk or run) to a position relative to the current position. */
    void MoveTo(float x,float y,float z);
    /** play an animation by animation name */
    void PlayAnim(const string& sAnim);
    /** play an animation by animation id */
    void PlayAnim(int nAnimID);
    /** wait the specified seconds, without further processing commands */
    void Wait(float fSeconds);
    /** execute a given script command*/
    void Exec(const string& sCmd);
    /** pause the sequence infinitely until some one resumes it. */
    void Pause();
    /** turn the character to face a given absolute direction in radian value. */
    void Turn(float fAngleAbsolute);
    /** move forward using the current facing a given distance*/
    void MoveForward(float fDistance);
    /** move backward using the current facing a given distance*/
    void MoveBack(float fDistance);
    /** move left using the current facing a given distance*/
    void MoveLeft(float fDistance);
    /** move right using the current facing a given distance*/
    void MoveRight(float fDistance);

```

```

/** Jump once */
void Jump();
/** offset the current sequence commands by a given steps.*/
void Goto(int nOffset);
/** offset to a label, if label not found, it will wrap to the beginning.*/
void Goto(const string& sLabel);
/** add a new label at the current position.*/
void Lable(const string& sLabel);

private:
    SequenceEntity* m_pSequenceEntity;

    enum SEQ_STATE{
        SEQ_EMPTY, // empty
        SEQ_CREATED, // only resides in memory
        SEQ_MANAGED, // is being managed by the manager
    }m_nSequenceState;
    /** sequence name */
    string      m_name;
    /** description */
    string      m_description;
    /** whether how sequence should be played.*/
    int m_nPlayMode;
    /** initial position in world coordinate */
    D3DXVECTOR3   m_vStartPos;
    /** initial facing */
    float       m_fStartFacing;
    int m_nKeyPos;
    bool m_bIsAddingKey;
    /** whether is playing forward*/
    bool m_bForward;
    /** 1/FPS, where FPS is the frequency of valid activation.*/
    float m_fMinInterval;
    float m_fUnprocessedTime;
    /** the number seconds that a sequence item has been executed.*/
    float m_fItemDuration;
    /** the number of seconds after which all move commands will be treated as finished.
    default value is 30 seconds.*/
    float m_fMovingTimeOut;
    /** default to 1 seconds */
    float m_fTurningTimeOut;

    /** copy the entity parameter to this controller.*/
    void CopyEntityParamter(const SequenceEntity& e);
    /** assign a new sequence to replace the current one.*/
    void SetEntity(SequenceEntity* e);
};


```

Implementation

```

int CSequenceCtler::AdvanceKey(int nOffset)
{
    int nCount = m_pSequenceEntity->GetTotalKeys();
    if(nCount == 0)
        return 0;
    int nPos = m_nKeyPos+nOffset;
    if(nPos<nCount && nPos>=0)
        m_nKeyPos = nPos;
    else

```

```

    {
        switch(m_nPlayMode)
        {
            case SequenceEntity::PLAYMODE_FORWARD:
            case SequenceEntity::PLAYMODE_BACKWORD:
                if(nPos >= nCount)
                {
                    nOffset = nCount-m_nKeyPos-1;
                    m_nKeyPos = nCount-1;
                }
                else // if(nPos<0)
                {
                    nOffset = m_nKeyPos;
                    m_nKeyPos = 0;
                }
                break;
            case SequenceEntity::PLAYMODE_FORWARD_LOOPED:
            case SequenceEntity::PLAYMODE_BACKWORD_LOOPED:
                if(nPos >= nCount)
                {
                    nPos = nPos%nCount;
                    m_nKeyPos = nPos;
                }
                else // if(nPos<0)
                {
                    nPos = nCount - (-nPos)%nCount;
                    nOffset = nPos;
                }
                break;
            case SequenceEntity::PLAYMODE_ZIGZAG:
                if(nPos >= nCount)
                {
                    m_bForward = false;
                    nPos = nPos%nCount;
                    m_nKeyPos = nCount - nPos;
                }
                else // if(nPos<0)
                {
                    m_bForward = true;
                    nPos = (-nPos)%nCount;
                    nOffset = nPos;
                }
                break;
            default:
                break;
        }
    }
    assert(m_nKeyPos>=0 && m_nKeyPos<nCount);
    return nOffset;
}
void CSequenceCtler::FrameMove(float fDeltaTime)
{
    if(IsActive())
    {
        CBipedObject* pBiped = GetBiped();
        CBipedStateManager* pState = NULL;
        if(pBiped==NULL || (pState=pBiped->GetBipedStateManager())==NULL)
            return;
        // process FPS.
    }
}

```

```

m_fUnprocessedTime += fDeltaTime;
fDeltaTime = m_fUnprocessedTime;
m_fItemDuration += fDeltaTime;
if(m_fUnprocessedTime > m_fMinInterval)
    m_fUnprocessedTime = 0;
else
    return;
bool bFinished = false;

///////////////////////////////
//
// check if the current sequence command is finished
//
/////////////////////////////
SequenceEntity::SequenceItem& item = m_pSequenceEntity->GetItem(m_nKeyPos);
int nOffset = m_bForward?1:-1;
switch(item.m_commandtype)
{
case SequenceEntity::CMD_MoveTo:
case SequenceEntity::CMD_WalkTo:
case SequenceEntity::CMD_RunTo:
case SequenceEntity::CMD_MoveForward:
case SequenceEntity::CMD_MoveBack:
case SequenceEntity::CMD_MoveLeft:
case SequenceEntity::CMD_MoveRight:
{
    if(m_fItemDuration > m_fMovingTimeOut)
    {
        bFinished = true;
    }
    else
    {
        D3DXVECTOR3 vCurPos = pBiped->GetPosition();
        D3DXVECTOR3 vDest = item.m_vPos+m_vStartPos;
        if (CMath::CompareXZ(vCurPos, vDest, 0.01f))
        {
            bFinished = true;
        }
        else if(pBiped->IsStanding())
        {
            nOffset = 0;
            bFinished = true;
        }
    }
    break;
}
case SequenceEntity::CMD_Turn:
{
    if(m_fItemDuration > m_fTurningTimeOut)
    {
        bFinished = true;
    }
    else
    {
        float fCurFacing = pBiped->GetFacing();
        float fFacing = item.m_fFacing + m_fStartFacing;
        if (abs(fFacing-fCurFacing)<0.01f)
        {
            bFinished = true;
        }
    }
}
}

```

```

        }

    }

    break;
}

case SequenceEntity::CMD_Wait:
{
    bFinished = (m_fItemDuration >= item.m_fWaitSeconds);
    break;
}

default:
{
    bFinished = true;
    break;
}

if(bFinished)
{
    /////////////////
    //
    // process the next sequence command
    //
    ///////////////
    m_fItemDuration = 0;
    bool bEndOfSequence = AdvanceKey(nOffset) != nOffset;

    if(bEndOfSequence)
    {
        Suspend();
        return;
    }

    SequenceEntity::SequenceItem& item = m_pSequenceEntity->GetItem(m_nKeyPos);
    D3DXVECTOR3 vDest = item.m_vPos+m_vStartPos;
    float fFacing = item.m_fFacing;
    switch(item.m_commandtype)
    {
        case SequenceEntity::CMD_MoveTo:
        case SequenceEntity::CMD_WalkTo:
        case SequenceEntity::CMD_RunTo:
        case SequenceEntity::CMD_MoveForward:
        case SequenceEntity::CMD_MoveBack:
        case SequenceEntity::CMD_MoveLeft:
        case SequenceEntity::CMD_MoveRight:
            pState->SetPos(vDest);
            pState->SetAngleDelta(fFacing);
            if(item.m_commandtype == SequenceEntity::CMD_WalkTo)
                pState->SetWalkOrRun(true);
            else if(item.m_commandtype == SequenceEntity::CMD_RunTo)
                pState->SetWalkOrRun(false);

            pState->AddAction(CBipedStateManager::S_WALK_POINT, (const void*)1);
            break;
        case SequenceEntity::CMD_Turn:
        {
            pBiped->FacingTarget(fFacing);
            break;
        }
        case SequenceEntity::CMD_Exec:
        {
    }
}

```

```

        string sFile, sCode;
        DevideString(item.m_strParam, sFile, sCode, ';');
        CGlobals::GetScene()->GetScripts().AddScript(sFile, 0, sCode, pBiped);
        break;
    }
    case SequenceEntity::CMD_Pause:
    {
        Suspend();
        break;
    }
    case SequenceEntity::CMD_PlayAnim:
    {
        if(!item.m_strParam.empty())
            pState->AddAction(CBipedStateManager::S_ACTIONKEY,
&ActionKey(item.m_strParam));
        else
            pState->AddAction(CBipedStateManager::S_ACTIONKEY,
&ActionKey(item.m_dwValue));

        break;
    }
    case SequenceEntity::CMD_Jump:
    {
        pState->AddAction(CBipedStateManager::S_JUMP_START);
        break;
    }
    case SequenceEntity::CMD_Goto:
    {
        if(!item.m_strParam.empty())
        {
            // m_nGotoOffset is absolute position for goto command with Labels, so ...
            AdvanceKey(item.m_nGotoOffset - m_nKeyPos);
        }
        else if(AdvanceKey(item.m_nGotoOffset) != item.m_nGotoOffset)
            Suspend();
        break;
    }
    default:
        break;
    }
}
}

```

14.4 Summary and Outlook

In this chapter, we have examined how game AI can be implemented from a game engine's perspective. AI is a very versatile and broad topic and it worth 3 books like this to write about in details. Despite of this, this chapter tries to sort out most practical AI techniques used in the RPG game genre and described a valid and common AI framework, where pieces of specific AI logics can fit together. The task of AI framework in a game engine is to provide means for designers and programmers to add game specific AI logics after the game engine is released, as well as integrating custom AI modules with other modules in the game engine, such as physics, animations, etc.

Obviously there are quite a number of AI techniques that are not covered in this chapter. Some of them can be found in third-party middleware as well, such as procedural animation, path-finding, flocking and crowd simulation, HFSM (hierarchical finite-state machine) editors, and scripting systems, etc.

However, it should be pointed out that any game AI is built on top of some underlying engine technology, and is built in the context of a specific game design. If either one of these things changes significantly, it may break the AI framework completely.

On the tech side, if one changes the character simulation, the game object representation, or the animation system significantly, one may well have to modify, if not rewrite it from scratch, significant portion of AI code. Similarly, on the design side, a huge amount of work in AI involves tuning and tweaking it for a specific game. The real challenge in game AI is customizing it to be a perfect fit for a certain game design. Game engine developers need to be aware of this.

In the long run, we can hope for a unified AI framework which could deal with all virtual and real world situations. The invention of computers brings AI; and it is all because of AI that we use a computer. AI in computer games is about human-like AI, which is most challenging and fascinating to us.

Chapter 15 Navigating in 3D world

In previous chapters, we have shown how a 3D world can be modeled, rendered and simulated. This chapter focuses on navigation in the 3D game world. In most games, camera system is the primary navigation tool for players. Game characters in the scene also need their own navigation systems to travel in the world either autonomously or through some predefined routes. This chapter is organized in several navigation related topics.

15.1 Camera Control

The major view into the 3D game world is controlled by the game engine's camera control system. The main task of a camera control system is positioning the camera in the 3D world to provide a good view of some relevant 3D objects. In a game engine, we usually define a good view as a set of measurable constraints, some of which are defined by game rules and others are related to the physical environment. For example, we may require that the camera having an unobstructed view of the main player in the 3D environment.

15.1.1 Background

Automatically planning camera shots in 3D virtual environments requires solving problems similar to those faced by human cinematographers. In the most essential terms, each shot must communicate some specified visual message or goal. Consequently, the camera must be carefully staged to clearly view the relevant subject(s), properly emphasize the important elements in the shot, and compose an engaging image that holds the viewer's attention. The constraint-based approach to camera planning in 3D virtual environments is built upon the assumption that camera shots are composed to communicate a specified visual message expressed in the form of constraints on how subjects appear in the frame. An external module issues a request to visualize some given subjects and specifies how each should be viewed; then a constraint solver attempts to find a camera shot solution which satisfies these constraints, including all camera parameters such as position and orientation.

For example, there are constraints for different camera modes, such as third person, first person, birds view, etc. There are constraints for certain scene objects, such as a camera frustum must either stay under water or above water, but not in the middle. There are constraints that force the camera to move along some predefined route. One of the most important camera constraints in game, however, is occlusion constraint.

15.1.2 Occlusion Constraint

Occlusion constraint is about reposition the camera so that the camera view frustum stays in free space and there are no opaque physical objects between the camera eye position and the look at point. We have seen some games that make object meshes transparent when they intersect with the camera view frustum or obstruct the camera view. In fact, this can be our last resolve if there are no solutions by simply reposition the camera in the scene. Below is the occlusion solver implementation. Please note that it requires that the physics engine to provide the query results.

Occlusion constraint is usually applied after all other camera constraints are applied. The output from the previous constraint solver becomes the input of the occlusion constraint solver.

Input:

- vEye: desired camera eye position
- vLookAt: desired camera look at position

Goal:

Adjust vEye and vLookat, so that:

- (1) There are no physical objects in between
- (2) The near plane of the camera view frustum does not collide with any mesh object or the terrain

Solver for Goal (1): Line of Sight Solver

Cast a ray from the desired look at position to desired eye position. If the ray hits any physical object, let fLineOfSightLen be the distance from the intersection point to the ray origin. If the ray hits nothing, fLineOfSightLen is infinity. The output camera parameters are further determined using the value of fLineOfSightLen as below.

- If fLineOfSightLen is larger than the line of sight with the desired camera position, then the desired eye position and look at position are adopted.
- If fLineOfSightLen is smaller than the distance from the near camera plane to the camera eye; the camera position does not change (both eye and look at location remain unchanged).
- If fLineOfSightLen is smaller than the line of sight with the desired camera position, but larger than the distance from the near camera plane to the camera eye, the desired look at position is adopted, whereas the camera eye position is changed to the interaction point, and the camera eye movement speed is set to infinity

Solver for Goal (2): View Frustum Solver

For all kinds of cameras, we ensure that the camera's near plane rectangular and the eye position are well above the terrain surface. This is done by casting five rays from the five points (four are the corners of the near plane, one is the camera eye) downwards (along the negative Y axis) to the terrain surface. We will translate the camera eye position upward until all ray-terrain distances are positive. A similar method is used to avoid camera's near plane from intersecting with the physical mesh. This is done by casting a ray from the center of the near plane downward to see if it intersects with the physics mesh. Finally, we shift both the camera eye and camera look at position along the positive Y axis (world up) for some small distance calculated during the previous step.

Figure 15.1 shows an indoor camera automatically adjusted to provide an unobstructed view of the player in control.



Figure 15.1 Camera Occlusion Constraint and Physics

15.1.2.1 Code

Here is the code snippet for occlusion constraint.

```
// solver 1
{
    // the vector from the new look at position to the new camera eye position.
    D3DXVECTOR3 vReverseLineOfSight = vEye-vLookAt;
    float fDesiredLineOfSightLen = D3DXVec3Length(&vReverseLineOfSight);
    float fMinLineOfSightLen = m_fNearPlane+fCharRadius;
    if(fDesiredLineOfSightLen < fMinLineOfSightLen )
        fDesiredLineOfSightLen = fMinLineOfSightLen;

    D3DXVec3Normalize(&vReverseLineOfSight, &vReverseLineOfSight);
    D3DXVECTOR3 vHitPoint;
    float fLineOfSightLen;
    fLineOfSightLen = CGlobals::GetScene()->Pick(vLookAt, vReverseLineOfSight, NULL,
&vHitPoint, false);
    if((fLineOfSightLen >= fDesiredLineOfSightLen) || (fLineOfSightLen<0))
    {
        // the new vEye, vLookAt is adopted.
    }
    else if (fLineOfSightLen<=fMinLineOfSightLen)
    {
        // restore to old value, nothing is changed.
        vEye = m_vEye;
        vLookAt = m_vLookAt;
    }
    else
    {
        // use the (hit point - NearPlane) as the new eye position.
        vEye = vLookAt + vReverseLineOfSight*(fLineOfSightLen-m_fNearPlane);
        // Increase the camera transition speed.
        fEyeSpeed = m_fEyeSpeed * 5.f;
        fLookAtSpeed = m_fLookAtSpeed * 10.f;
    }
    /** the following code implements smooth roll back of the camera.
    */
    if(pBiped!=NULL)
    {
        float fCameraObjectDist = D3DXVec3Length(&(vLookAt-vEye));
    }
}
```

```

// 1000.f is a large value, beyond which speed rollback is disabled.
// 0.1f is minimum allowable object jerk distance. object smaller than this size will not cause
smooth rollback animation to apply.
    if(m_nForceNoRollbackFrames==0 && m_fCameraRollbackSpeed<1000.f &&
       ( fCameraObjectDist >
m_fLastCameraObjectDistance+max(m_fCameraRollbackSpeed*fElapsedTime, 0.1f)))
    {
        float fDist = m_fLastCameraObjectDistance + m_fCameraRollbackSpeed*fElapsedTime;
        vEye = vLookAt + (vEye-vLookAt)/fCameraObjectDist*fDist;
        m_fLastCameraObjectDistance = fDist;
    }
    else
        m_fLastCameraObjectDistance = fCameraObjectDist;
}
}

// solver 2
{
    /**
     * check for global terrain.
     */
    // get the view matrix
    D3DXMATRIX MatView;
    D3DXMatrixLookAtLH( &MatView, &(m_vEye), &(m_vLookAt), &vWorldUp );
    D3DXMATRIX* pMatProj = GetProjMatrix();

    // get the four points on the near frustum plane
    D3DXMATRIX mat;
    D3DXMatrixMultiply( &mat, &MatView, pMatProj );
    D3DXMatrixInverse( &mat, NULL, &mat );

    D3DXVECTOR3 vecFrustum[6];
    vecFrustum[0] = D3DXVECTOR3(-1.0f, -1.0f, 0.0f); // xyz
    vecFrustum[1] = D3DXVECTOR3( 1.0f, -1.0f, 0.0f); // Xyz
    vecFrustum[2] = D3DXVECTOR3(-1.0f, 1.0f, 0.0f); // xYz
    vecFrustum[3] = D3DXVECTOR3( 1.0f, 1.0f, 0.0f); // XYz
    vecFrustum[5] = m_vEye;

    for( int i = 0; i < 4; i++ )
        D3DXVec3TransformCoord( &vecFrustum[i], &vecFrustum[i], &mat );

    // one additional point (at the near plane bottom)to add more accuracy
    vecFrustum[4] = (vecFrustum[0]+vecFrustum[1])/2;

    float fShiftHeight = -EPSILON;
    // for all five point, test
    for( int i = 0; i < 6; i++ )
    {
        float fMinHeight = CGlobals::GetGlobalTerrain()->GetElevation(vecFrustum[i].x,
vecFrustum[i].z);
        if( fShiftHeight < (fMinHeight-vecFrustum[i].y))
            fShiftHeight = fMinHeight-vecFrustum[i].y;
    }
    /// ignore camera collision with the terrain object, if the camera is well below the terrain
surface.(may be in a cave or something)
    /// 2 meters is just an arbitrary value.
    if(fShiftHeight > 2.0f)
        fShiftHeight = -EPSILON;
}

/**

```

```

* check for physical meshes.
*/
{
    D3DXVECTOR3 vHitPoint, vPt;
    // the distance to check is m_fNearPlane*2, which is far larger than the near plane height
    /// we will check three points around the near plane.
    vPt=(vecFrustum[0]+vecFrustum[3])/2;
    float fDist = CGlobals::GetScene()->Pick(vPt, D3DXVECTOR3(0,-1,0), NULL, &vHitPoint, false,
m_fNearPlane*2);
    vPt = (vecFrustum[0]+vecFrustum[2])/2;
    float fDistTmp = CGlobals::GetScene()->Pick(vPt, D3DXVECTOR3(0,-1,0), NULL, &vHitPoint,
false, m_fNearPlane*2);
    fDist = min(fDistTmp, fDist);
    vPt = (vecFrustum[1]+vecFrustum[3])/2;
    fDistTmp = CGlobals::GetScene()->Pick(vPt, D3DXVECTOR3(0,-1,0), NULL, &vHitPoint, false,
m_fNearPlane*2);
    fDist = min(fDistTmp, fDist);

    if(fDist>0)
    {
        float fNearPlaneHeight = D3DXVec3Length(&(vecFrustum[2]-vecFrustum[0]));
        float fShift = fNearPlaneHeight/2-fDist+0.1f;
        if(fShift > fShiftHeight)
        {
            fShiftHeight = fShift;
        }
    }
}

/**
 * shift the distance
 */
if(fShiftHeight > -EPSILON)
{
    m_vEye.y+=fShiftHeight;
    m_vLookAt.y+=fShiftHeight;
}

```

15.2 Generic Path Finding Algorithms

Path finding is very common in video games. Few things make a game look "dumber" than bad path finding. Fortunately, path finding is mostly a "solved" problem. A* and its extensions are popular algorithms, which can efficiently build optimal paths between two endpoints, even if there is a long distance and many obstacles in the middle.

Two useful path finding algorithms in games are (1) simple wall sliding and (2) A* algorithm. We have already implemented wall sliding in the simulation Chapter 8. A* path finding is not very suitable for outdoor games with extensive and dynamic maps. Instead we will look at some other general path-finding solutions based on simple geometries.

15.2.1 Object Level Path Finding

Object level path-finding is path-finding in the 3D world without taking the polygon-level shape of objects in to consideration. It is mostly used for NPC character navigations. In

ParaEngine, object level path-finding will use the result generated by the character and environment simulator. Path-finding function is implemented by each biped object.

The available information for path-finding includes the terrain tile that the character belongs to and the distance to all other characters and obstacles in its perceptive radius. The goal of path-finding is to generate additional waypoints to the desired destination. There are several kinds of waypoints that a path-finding algorithm can generate for a character. See Table 15.1.

Table 15.1 Waypoint type

```
enum WayPointType {
    /// the player must arrive at this point, before proceeding to the next waypoint
    COMMAND_POINT=0,
    /// the player is turning to a new facing.
    COMMAND_FACING,
    /// when player is blocked, it may itself generate some path-finding points in
    /// order to reach the next COMMAND_POINT.
    PATHFINDING_POINT,
    /// The player is blocked, and needs to wait fTimeLeft in order to proceed.
    /// it may continue to be blocked, or walk along. Player in blocked state, does not have
    /// a speed, so it can perform other static actions while in blocked mode.
    BLOCKED
};
```

The waypoint generation rules are given in Table 15.2.

Table 15.2 Path-finding rule

rule1: we will not prevent any collision; instead, path-finding is used only when there are already collisions between this biped and the environment by using the shortest path.

rule2: if there have been collisions, we will see whether we have already given solutions in previous path-finding processes. If so, we will not generate new ones.

rule 3: we will only generate a solution when the next way point is a command type point.

A path finding solution can be generated as follow: when several objects collide with each other, only one object is selected to implement path-finding, while others are put to a blocked state. The selected object will pick the biggest object that collides with it and try to side-step it by generating additional waypoints between its current location and the original destination. See Figure 15.2

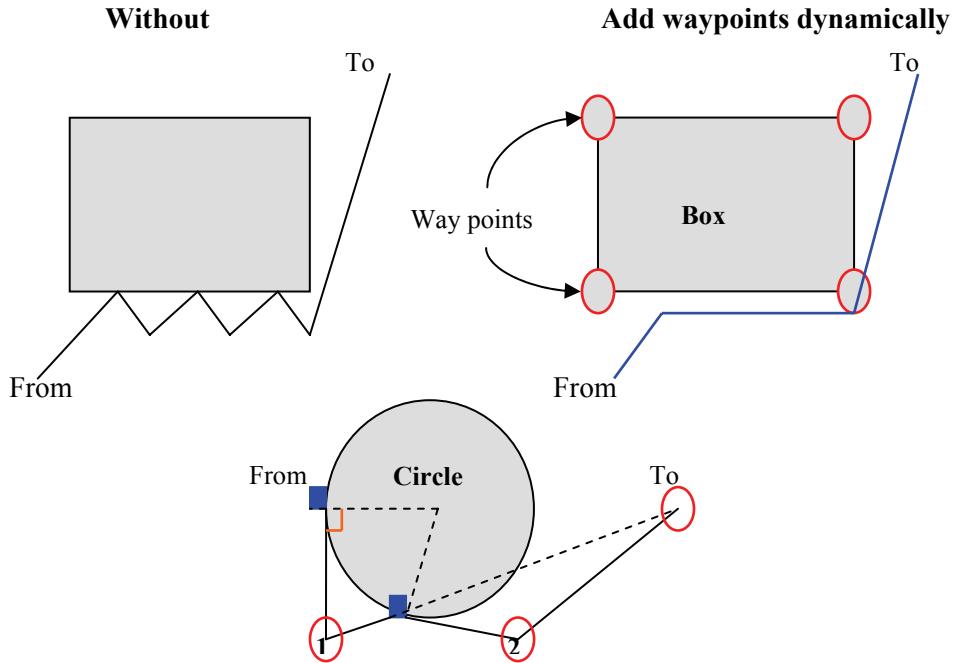


Figure 15.2 path-finding: adding dynamic waypoints

15.2.2 Path-finding and Physics

When path-finding and physics have conflicts, the physics always has the priority.

Game physics is calculated in real coordinate system, if path-finding is also done in real coordinate system, things can get very complicated. For example, the player can be at any position in the set $\{x,y,z\}$, where x,y,z are real numbers. Object collisions therefore can not be simply denoted by two states 0, 1; instead some fuzzy function $\text{DepthOf}(\text{object1}, \text{object2})$ is used to return the collision depth as a real number between 0 and 1 inclusive. The larger the value, the deeper the two objects run into each other. This is why graph-based path-finding algorithm (like A*) can not be applied directly in real-coordinate system based 3D environment.

To verify a solution generated by a real coordinate system based path-finding algorithm, we need to ensure that the sum of all collision depth values between each pair of collided objects decrease by at least some fixed value every frame.

15.2.3 Code

Since path-finding in real coordinate system involves tweaks for a specific game. We will only show the top level path-finding code that we implemented for biped object in our game.

```
//-----
/// desc: this function is called by the environment simulator, when needed.
/// The biped object then get an opportunity to shun from any obstacles or
/// try to solve collisions by generating additional way points.
```

```

//-----
void CBipedObject::PathFinding(double dTimeDelta)
{
#ifndef TURNOFF_BIPED_COLLISION
    return;
#endif
    /// rule1: we will not foresee any further, but implement path-finding only when
    /// there is already collisions between this biped and the environment.
    if(GetNumOfPerceivedObject()==0)
        return;

    BipedWayPoint waypoint;
    D3DXVECTOR3 vSrcPos;
    GetPosition(&vSrcPos);

    while(GetWayPoint(&waypoint))
    {
        /// rule2: if there have been collisions in the way, we will see whether we have already given
        /// solutions in the previous path-finding process. If so, we will not generate new ones nor canceling
        it

        if((waypoint.GetPointType() == BipedWayPoint::PATHFINDING_POINT) ||
           (waypoint.GetPointType() == BipedWayPoint::BLOCKED))
            return;

        /// rule 3: we will only generate a solution when the next way point is a command type point.
        if(waypoint.GetPointType() == BipedWayPoint::COMMAND_POINT )
        {
            // source and destination points
            D3DXVECTOR3 vDestPos = waypoint.vPos;

            {
                /**
                 * already reached destination, we will go on to the next way point in the queue
                 * when there is collision, we will regard a circle(with a radius) as the destination point.
                 * while in free-collision condition, a destination is really a point.
                 */
                float fLength = D3DXVec3LengthSq(&(vDestPos-vSrcPos));
                if(fLength<=(dTimeDelta*GetAbsoluteSpeed())*(dTimeDelta*GetAbsoluteSpeed()) ||
                   fLength<=(GetPhysicsRadius()*GetPhysicsRadius()))
                {
                    RemoveWayPoint();
                    continue;
                }
            }

            /**
             * See whether there are bipeds that are moving. If so, block the current biped for some seconds
             * or if the destination point(radius 0.01f)is inside one of the object, remove the way point
             * Get the biggest non-mobile object
             */
            bool bReachedObject = false;
            bool bShouldBlock = false;
            CBaseObject* objBiggest = NULL;
            float fBiggestRadius = 0.f;
            {
                float fMyRadius = GetPhysicsRadius();

                int nNumPerceived = GetNumOfPerceivedObject();

```

```

for(int i=0;i<nNumPerceived;++i)
{
    IGameObject* pObj = GetPerceivedObject(i);
    if(pObj!=NULL)
    {
        if(pObj->TestCollisionSphere(&vDestPos, fMyRadius/*0.01f*OBJ_UNIT*/, 0))
        {
            bReachedObject = true;
            break;
        }
        if( ! pObj->IsStanding() )
            bShouldBlock = true;
        if(!bShouldBlock && fBiggestRadius<= pObj->GetPhysicsRadius())
        {
            fBiggestRadius = pObj->GetPhysicsRadius();
            objBiggest = pObj;
        }
    }
}
if(bReachedObject == true)
{
    RemoveWayPoint();
    continue;
}
else if(bShouldBlock)
{
    // TODO: set a reasonable value. Just wait 0.5 seconds
    AddWayPoint(BipedWayPoint(0.5f));
    continue; // actually this is equivalent to return;
}

/**
 * We will only solve against the biggest static object, we can give very precise solution
 * according to its shape, when there is only one object.
 */
if(objBiggest)
{
    OneObstaclePathFinding(objBiggest);
    break;
}
else if(waypoint.GetPointType() == BipedWayPoint::COMMAND_MOVING)
{
    CBaseObject* objBiggest = NULL;
    float fBiggestRadius = 0.f;
    {
        float fMyRadius = GetPhysicsRadius();
        int nNumPerceived = GetNumOfPerceivedObject();
        for(int i=0;i<nNumPerceived;++i)
        {
            IGameObject* pObj = GetPerceivedObject(i);
            if(pObj!=NULL)
            {
                if(fBiggestRadius<= pObj->GetPhysicsRadius())
                {
                    fBiggestRadius = pObj->GetPhysicsRadius();
                    objBiggest = pObj;
                }
            }
        }
    }
}

```

```

        }
    }
}
if(objBiggest)
{
    OneObstaclePathFinding(objBiggest);
    break;
}
else
{
    return;
}
}
}

```

15.3 Summary and Outlook

In 3D game world, there are three levels of navigation. This chapter talks about the first two.

- Polygon level navigation: such as the camera controller and character's collision avoidance with mesh objects.
- Object level navigation: such as NPC character path-finding.
- World level navigation: such as exploring new regions of the virtual world on the Internet.

3D Navigation is also a good research topic. It involves not only algorithms, but also human computer interface (HCI) design and other psychological studies. As the game world gets bigger and bigger, rich HCI enabled navigation design and search technologies will become necessary. Aids objects such as symbols and marks can also be very helpful both for agent and human user. In fact, these things are already introduced in some latest adventure and MMORPG games.

One of the greatest advantages about things going 3D is immersion or the player's feel of being there. Yet one of the greatest disadvantages of going 3D is poor navigation compared to its 2D counter part. From the HCI perspective, this is mostly caused by the input devices that we used i.e. the mouse and the key board. If navigation in 3D games can be improved even slightly, it will give players a much more engaging gaming experience.

Chapter 16 NPL Scripting System

Scripts are external programs loaded and executed by a game engine at runtime. It allows game logics to be composed outside the hard-core of a game engine. The original idea behind a custom scripting language for a game is that one does not have to hard-code every single possible event and conversation into the game executable. It also avoids the need to recompile the executable for the game each time one wishes to slightly alter the storyline. Modern game engines natively support this scripting paradigm, which allows game developers to extend the original scripting language according to specific game needs.

In recently released computer games, most game contents are written in script files; these include GUI, game settings, event handlers, AI, and many other game logics. The scripting system in ParaEngine is called NPL scripting system. We build the entire scripting system based on some network infrastructure, so that script files does not necessarily run on a single client application, but may be deployed and run by multiple applications on the network. This makes writing and deploying networked computer games much easier and much more flexible.

In this chapter, we will mainly focus on client side scripting system and its implementations. In the second volume of this book, we will focus on server technologies using the NPL scripting system.

16.1 Foundation

Building a scripting system by extending an existing language is a great short cut to immediately add scripting capabilities to your game engine in a matter of hours. There are several available base languages that one can choose from, such as Lua, Python, Java, and C#. Each of these languages has a number of successful games using it.

We suggest Lua, which is a hidden secret in the game community for a decade. Nowadays, it receives wide popularity among both large game studios and indie game developers. One will find rich community support on script editors, debuggers, wrappers for database, and many script examples from released games, etc.

In ParaEngine, NPL scripting system is based on Lua, a popular light-weighted extensible language. It offers good support for object-oriented programming, functional programming, and data-driven programming. This chapter is not about telling you how to integrate Lua into your game engine. The Lua documentation and the community website have already done a good job on that. Instead, we are going to show you how to use the scripting paradigm most effectively in a game engine. Since the rest of the chapter is mainly written for developers, we assume that you have already read all the Lua documentation carefully.

16.1.1 Using LuaBind

LuaBind is a set of C++ templates, which makes binding C++ functions to Lua very easy. We have used it extensively in ParaEngine, except for a few occasions where their grammar involves flexible parameter passing.

In the game engine, we build an exact C++ copy of everything that is exposed to the scripting interface and document them in the header files. Then we use LuaBind to bind these

C++ functions and classes to Lua tables. We do not advice to directly expose C++ objects which are used by the game engine to the scripting interface, because it will pollute the C++ game engine code and later become very difficult to separate code and documentations for the script and for the game engine. The following shows the example.

Suppose we want to expose a game engine object CBaseObject to the scripting interface. Instead of using CBaseObject directly, we can create another dedicated wrapper class called ParaObject for exportation. ParaObject contains a pointer to CBaseObject as well as a set of wrapper functions of CBaseObject. Please note that we document ParaObject as if it is a script object. See below.

```
/*
 * @ingroup ParaScene
 * ParaObject class:
 * it is used to control game scene objects from scripts.
 @par Class Properties

 - ("name",&ParaObject::GetName,&ParaObject::SetName)
 */

struct ParaObject
{
public:
    CBaseObject* m_pObj;      // a pointer to the object

    ParaObject():m_pObj(NULL){};
    ParaObject(CBaseObject* pObj):m_pObj(pObj){};
    ~ParaObject(){};

    /** get the Runtime class information of the object. */
    string GetType() const;
    /** get paraengine defined object type name.
     // TODO: This function is not implemented
    */
    int GetMyType() const;
    /**
     * check if the object is valid
    */
    bool IsValid() const {return m_pObj!=NULL;};
    /** whether the object has been attached to the scene. */
    bool IsAttached() const;

    /** get the attribute object associated with an object. */
    ParaAttributeObject GetAttributeObject();

    /** Rotate the object. This only takes effects on objects having 3D orientation, such as
     * static mesh and physics mesh. The orientation is computed in the following way: first rotate
     around
     * x axis, then around y, finally z axis.
     * Note: this function is safe to call for all kind of objects except the physics mesh object.
     * for physics mesh object, one must call ParaScene.Attach() immediately after this function.
     * for more information, please see SetPostion();
     * @param x: rotation around the x axis.
     * @param y: rotation around the y axis.
     * @param z: rotation around the z axis.
     * @see: SetPostion();
    */
    void Rotate(float x, float y, float z);
}
```

```

        // ... many wrapper functions are ignored here
    }
}

```

Using LuaBind, we can easily bind the above class object to the scripting interface as below. After running the binding code, we will be able to use ParaObject from script files.

```

module(L)
[
    namespace_("ParaScene")
    [
        // ParaObject class declarations
        class_<ParaObject>"ParaObject"
            .def(constructor<>())
            .property("name", &ParaObject::GetName, &ParaObject::SetName)
            .def("IsValid", &ParaObject::IsValid)
            .def("GetType", &ParaObject::GetType)
            .def("GetMyType", &ParaObject::GetMyType)
            .def("IsAttached", &ParaObject::IsAttached)
            .def("GetAttributeObject", &ParaObject::GetAttributeObject)
            .def("Rotate", &ParaObject::Rotate)
        // ... many declarations ignored here
    ];
]

```

The above code does not look like C++ code, but in fact it is. One can debug into it to see how LuaBind generates the registration code at the compile time and run time.

In Lua script file, we can use the ParaObject as below.

```

-- suppose Obj is a ParaObject
if( Obj:isValid() == true ) then
    Obj:Rotate(30, 0, 0);
end

```

The above programming paradigm provides us a centralized way to expose C++ objects to the scripting interface. In the next section, we will look at a decentralized way to expose C++ attributes to the scripting interface.

16.1.2 Exporting C++ Attributes to Script

In this section, we will look at another very automatic way to expose C++ object attributes to the scripting interface.

In interpreted language like C#, there is a concept called class reflection, which allows programmers to query an object's the class type information at runtime and execute functions found in the class type information. Class type information enables us to access an object's properties and methods without knowing its class type at compile time.

In C++, we can simulate this functionality with little code cost. Whenever we want to expose an attribute from a C++ class object such as the CBaseObject to the scripting interface, we just add the get and set methods of the attribute to the CBaseObject's class type information, and it is done. We do not need to register the attribute in Lua and we do not need to make a copy of the set and get methods in the dedicated wrapper class (or ParaObject in this case). The only disadvantage is some decreased performance and slightly increased code length when getting or setting an object's attributes in script files. Hence this method is suitable for exporting attributes and functions that are not called very frequently in script files.

In ParaEngine, we store class runtime information in an attribute object. All instances of a class share the same attribute object, which can be retrieved from a class instance by calling its `GetAttributeObject()` method; the attribute object can then be used to get and set attributes of the class instance.

An attribute object contains a mapping from each attribute name string to whatever is needed to manipulate that attribute given a class instance pointer. The implementation is given below. Please note that the implementation is not unique. Generally speaking, there are three different implementations based on three different ways to store the type information. Each has its advantages and disadvantages. We will look at them one by one.

16.1.2.1 Using Data Member Offset

In this method, the attribute object maps each attribute name string to a corresponding class data member offset value. We can get the offset of a class's data member using the following macro.

```
/** @def get the offset of a data member(x) from a class(cls) */
#define Member_Offset(x, cls) (int)&((cls *)0)->x
```

Given the pointer to a class instance, we can compute the memory address of the data member by adding the pointer to the member offset. This method requires the minimum effort to implement. However, it only works on data members, but not on member functions. Moreover, no data validation can be performed, so it is usually not safe to use.

16.1.2.2 Using Member Function Pointers

In this method, the attribute object maps each attribute name string to the corresponding class's member function pointer. Given the pointer to a class instance, we can call its member function by the coorepsonding member function pointer.

This sounds a very good idea. It is safe to use because class data is always set or get from class member functions, where data validation can be performed. However, C++ class member function pointer is not easy to store. There is a myth even among experienced C++ developers that member function pointers are merely some offset values like the class data member offset. Unfortunately, they are not that simple. Both STL and boost library provides some helper templates to ease the use of member functions in C++. One can refer to these places to find out why.

Implementing the attribute object functionalities using member function pointers requires a great deal of template programming in C++, and one may end up with the wrong result. Hence, we advise the third method which is given next.

16.1.2.3 Using Static Member Function Pointers

This is our approach in NPL scripting system. The attribute object maps each attribute name string to the corresponding class's static member function pointer. A static function uses *stdcall* calling convention and its function pointer has a fixed data length. Static class member function can simulate a member function by simply setting the first parameter to the class pointer. Take the `CBaseObject` for example; suppose we want to expose the `SetPosition` and `GetPosition` member functions of the `CBaseObject`, we just add two static functions as follow.

```

Class CBaseObject{
Public:
    static int GetPosition_s (CBaseObject* cls, D3DXVECTOR3* p1) {
        cls->GetPosition(p1); return S_OK;
    }
    static int SetPosition_s (CBaseObject* cls, D3DXVECTOR3 p1) {
        cls->SetPosition(&p1); return S_OK;
    }
// ... many code omitted
}

```

Given the pointer to a class instance, we just call the attribute's corresponding static function and pass the class instance pointer as the first parameter. Although this is cumbersome sometimes (programmers need to add two more lines of code in the header file for each property), it works well with overloaded functions, multiple inheritance and virtual functions. Most importantly: (1) Attribute object only needs to be built once (automatically built on first use), and is valid for all instances of the same class type thereafter. (2) It gives programmers individual control over the behaviors of each attribute. E.g. we can make the behavior of the attribute's set/get methods slightly different from the corresponding class member functions, such as adding more robust data validation and error reporting functionalities, or even implicitly changing the class pointer, so that we can expose attributes from both an object's class members and its base class. (3) It makes the executable file smaller compared with the implementation using template programming.

Please see the code section for implementation details.

16.1.3 Runtime State Management

When loading or executing a script file, we must specify the runtime environment in which the script code is loaded and executed. One can think of each runtime environment as an independent virtual machine. The Lua's runtime environment is fully reentrant: it has no global variables. The whole state of the Lua interpreter (global variables, stack, etc.) is stored in a dynamically allocated structure called Lua state. Lua's runtime environment is very light weighted, which allows us to create and host hundreds of those Lua states effectively. However, in most cases, we will load and execute script files in a shared global runtime state, which is created when the game engine starts. An alternative is to load and execute files in separate runtime states.

There are some discussions on whether to use one giant runtime state or many small light-weighted states. The design principle is usually not on performance, but on data sharing. In Lua, all functions and variables are data tables in a global environment. Hence script files loaded in the same runtime state can access all tables created in it. This is a very important feature, because otherwise the only way to share data is via the game engine. The following design principles can be helpful when developing your game.

- GUI and major game logics run in the same main runtime state.
- Most AI events run in the main runtime state, some may run in separate runtimes.
- Any unsafe and unauthorized script code, such as those entered by the users or from the network, may need to run in separate runtime states.

16.1.4 Using Namespace

The beauty of Lua is its flexibilities. However, without proper administration, we will quickly populate the main runtime state with tons of data and functions in a flat hierarchy. One solution to this problem is that we organize host API (which is exposed from the game engine to the script system. See Lua Documentation for details) into namespaces and encourage script writers to do the same for any class and data structures created in the script files.

In actual game development, several designers and programmers may jointly work on the thousands of script files. They have little idea how other people are going to name variables in their script files. Hence it is good practice to define a list of reserved global variables and a list of namespaces in which corresponding game modules should be developed. The following subsections show some examples.

16.1.4.1 Reserved Namespaces

The following shows an example of reserved namespaces for host API in ParaEngine. The host API exposed by ParaEngine is a group of API to construct 2D and 3D content.

Table Name	Description
ParaCamera	The camera controller.
ParaScene	ParaScene namespace contains a list of HAPI functions to create and modify scene objects in ParaEngine
ParaAsset	ParaAsset namespace contains a list of HAPI functions to manage resources(asset) used in game world composing, such as 3d models, textures, animations, sound, etc.
ParaUI	ParaUI namespace contains a list of HAPI functions to create user interface controls, such as windows, buttons, as well as event triggers.
ParaMisc	Contains miscellaneous functions.
NPL	Neural Parallel Language foundation interface is in this namespace.

16.1.4.2 Reserved Classes

The following are some reserved data types or class objects in NPL.

Object Name	Description
ParaObject	It is represent a scene object.
ParaCharacter	It is represent a character object.
ParaAssetObject	It represents an assert entity
ParaUIObject	It represents a GUI object

16.1.4.3 Reserved Variables

The following are some reserved data types or class objects in NPL.

Variable Name	Description
sensor_name	In an event handler, this variable contains the name of the callee.
self	Equivalent to <i>this</i> pointer in C++.
mouse_button	In a mouse event handler, it is mouse button that triggers the mouse.
mouse_x	In a mouse event handler, it is mouse x position.
mouse_y	In a mouse event handler, it is mouse y position.
keyboard_key	In a key event handler, it contains a list of key strokes.
keyboard_character	In a key event handler, it contains a string of readable characters.

16.1.5 Using Directory and Domain Name

How can a script file be found and referenced either by a script writer or by another script? The answer is directory path plus script file name. Indeed, directory is the most intuitive and convenient way to organize script files, especially when there are thousands of them. In NPL scripting system, we allow a domain name and some decorator symbols to optionally precede the directory path when specifying a script file. This gives us more flexibility when referencing script files from both local and remote NPL runtimes. Domain name together with decorator symbols can be used to specify the runtime environment of script files. In NPL, they are mainly used in networked programming environment. For scripting on the local computer, directory path alone is sufficient. We will cover the details of network programming using NPL in the next volume of the book.

The following shows how a script function NPL.load() is used to load another script file.

NPL.load(const char * filePath, bool bReload=false)

Load a new glia file (in the local environment) without running it. If the glia file is already loaded, it will not be loaded again. IMPORTANT: unlike other activation functions, this is something similar to "include" in C programming, the function is loaded where it is and returned to the original caller upon finish.

Parameters:

filePath: the local file path

bReload: if true, the file will be reloaded even if it is already loaded. Otherwise, the file will only be loaded if it is not loaded yet.

Returns:

Return the GliaFile reference.

Examples:

```
NPL.load("(gl)myworld/npc/creature0.lua", false);
```

16.1.6 Script File Activation Mechanism

This part can be optional in a scripting system. However, it is a very important component in NPL scripting system and is used extensively both for client side and distributed programming. So we will briefly introduce it anyway.

Script file is a very important unit of execution in NPL scripting system, and we allow each script file to optionally provide an activation function. One can think of it as the main() function in C/C++ application programs; yet it may be called many times and also exits very fast. To define an activation function in a script file, we use the NPL.this() command. It can take any previously defined script functions as its input. See the following declaration.

NPL. this (const functor funcActivate)

Associate a user defined function as the activation function of this file. Add the current file name to the __activate table. Create the activate table, if it does not exist.

Parameters:

funcActivate: the function pointer to the activation function. It can either be local or global.

After registration, we can call the file's activation function by simply providing the file path, where the NPL.activate() command is used. See below.

NPL.activate (const char * sNeuronFile, const char * code = NULL)

Activate the specified file. It can either be local or using the default or specified DNS. All these information is extracted from the sNeuronFile parameters.

Parameters:

sNeuronFile: Neuron File name for this neuron file. It may contain activation type, namespace or DNS server information. They should be specified in the following format:
[(g|gl|l) [namespace :] relative_path [@ DNSServerName]]

//... many documentation ignored

code: It is a chunk of authorized code that will be executed in the destination file before its activation is called. This code usually set the values of the destination's global variables.

A script code which contains an activation function may look like this.

File name: SampleScript.lua

```

local function activate()

if (state == nil) then --state is a global variable
    local playname = "abc";
    -- activate another script on remote machine.
    NPL.activate("ABC: /pol_intro.lua", "state=nil;");
    -- activate another script on local machine.
    NPL.activate((gl)polintro.lua","");
else
    ...
end

end

state={}; -- a global variable (table), usually for passing and share states among NPL files.
NPL.this (activate); --tell NPL which function is used as the activation function of this file.

```

16.2 Architecture and Code

If you are extending your scripting system over Lua, then there should be no problem on architecture issues. All you need is some code to register host API functions to the Lua runtime. Hence we combined the architecture and code to the same section in this chapter.

We will illustrate how to export class attributes using static member functions in this section.

16.2.1 Attribute Field

The following are structures to store a single attribute.

```

/** a list of all attribute type*/
enum ATTRIBUTE_FIELDTYPE
{
    // get(), set()
    FieldType_void,
    // get(int*) set(int)
    FieldType_Int,
    // get(bool*) set(bool)
    FieldType_Bool,
    // get(float*) set(float)
    FieldType_Float,
    // get(float*,float* ) set(float, float)
    FieldType_Float_Float,
    // get(float*,float*,float*) set(float, float, float)
    FieldType_Float_Float_Float,
    // get(int*) set(int)
    FieldType_Elem,
    // get(double*) set(double)
    FieldType_Double,
}

```

```

// get(D3DXVECTOR2*) set(D3DXVECTOR2)
FieldType_Vector2,
// get(D3DXVECTOR3*) set(D3DXVECTOR3)
FieldType_Vector3,
// get(D3DXVECTOR4*) set(D3DXVECTOR4)
FieldType_Vector4,
// get(const char**) set(const char*)
FieldType_String,
FieldType_Deprecated = 0xffffffff
};

/** for a single attribute field */
class CAttributeField
{
public:
    CAttributeField();
    ~CAttributeField();

public:
    union any_offset{
        void* ptr_fun;
        int offset_data;
    };
    any_offset m_offsetSetFunc;
    any_offset m_offsetGetFunc;

    /** field name: e.g. "base.position" */
    string      m_sFieldname;
    /** see ATTRIBUTE_FIELDTYPE */
    DWORD       m_type;

    /** additional schematics for describing the display format of the data. Different attribute type have
different schematics.
@see GetSimpleSchema() */
    string      m_sSchematics;
    /** a help string.*/
    string      m_sHelpString;

public:
    /**
     * get the field type as string
     */
    const char* GetTypeAsString();

    inline HRESULT Call(void* obj)
    {
        if(m_offsetSetFunc.ptr_fun!=0)
            return ((HRESULT (*) (void* obj))m_offsetSetFunc.ptr_fun)(obj);
        else if(m_offsetGetFunc.ptr_fun!=0)
            return ((HRESULT (*) (void* obj))m_offsetSetFunc.ptr_fun)(obj);
        else
            return E_FAIL;
    };
    template <class datatype>
    inline HRESULT Set(void* obj, datatype p1)
    {
        if(m_offsetSetFunc.ptr_fun!=0)
            return ((HRESULT (*) (void* obj, datatype p1))m_offsetSetFunc.ptr_fun)(obj, p1);
        else
            return E_FAIL;
    };
    template <class datatype>

```

```

inline HRESULT Get(void* obj, datatype* p1)
{
    if(m_offsetGetFunc.ptr_fun!=0)
        return ((HRESULT (*) (void* obj, datatype* p1))m_offsetGetFunc.ptr_fun)(obj, p1);
    else
        return E_FAIL;
};

// ... many other set/get function templates ignored.

public:
    enum SIMPLE_SCHEMA {
        SCHEMA_RGB = 0,
        SCHEMA_FILE,
        SCHEMA_SCRIPT,
        SCHEMA_INTEGER,
    };
    static const char* GetSimpleSchema(SIMPLE_SCHEMA schema);
    static const char* GetSimpleSchemaOfRGB(){return GetSimpleSchema(SCHEMA_RGB);}
    static const char* GetSimpleSchemaOfFile(){return GetSimpleSchema(SCHEMA_FILE);}
    static const char* GetSimpleSchemaOfScript(){return GetSimpleSchema(SCHEMA_SCRIPT);}
    static const char* GetSimpleSchemaOfInt(int nMin, int nMax);
    static const char* GetSimpleSchemaOfFloat(float nMin, float nMax);

    /**
     * parse the schema type from the schema string.
     */
    const char* GetSchematicsType();

    /**
     * parse the schema min max value from the schema string.
     * @return true if found min max.
     */
    bool CAttributeField::GetSchematicsMinMax(float& fMin, float& fMax);
};

```

16.2.2 Attribute Class

This is the attribute object returned when querying a class instance for class type information.

```

/** an attribute class is a collection of attribute fields. */
class CAttributeClass
{
public:
    CAttributeClass(int nClassID, const char* sClassName, const char* sClassDescription);
    ~CAttributeClass(){}
    enum Field_Order
    {
        Sort_By_Name,
        Sort_By_Category,
        Sort_By_InstallOrder,
    };
public:
    /** add a new field.
        @param sFieldname: field name
        @param Type: the field type. It may be one of the ATTRIBUTE_FIELDTYPE.
        @param offsetSetFunc: must be __stdcall function pointer or NULL. The function prototype should
        match that of the Type.
        @param offsetGetFunc: must be __stdcall function pointer or NULL. The function prototype should
        match that of the Type.
        @param sSchematics: a string or NULL. The string pattern should match that of the Type.
        @param sHelpString: a help string or NULL.
    */

```

```

@param bOverride: true to override existing field if any. This is usually set to true, so that inherited
class can override the fields installed previously by the base class.
*/
void AddField(const char* sFieldname,DWORD Type, void* offsetSetFunc,void* offsetGetFunc,
const char* sSchematics, const char* sHelpString,bool bOverride);
/** use of deprecated field takes no effect and will output warning in the log. */
void AddField_Deprecated(const char *fieldName,bool bOverride=true);
/** remove a field, return true if moved. false if field not found. */
bool RemoveField(const char* sFieldname);
void RemoveAllFields();

/** class ID */
int GetClassID() const;
/** class name */
const char* GetClassName() const;
/** class description */
const char* GetClassDescription() const;

/** Set which order fields are saved. */
void SetOrder(Field_Order order);
/** get which order fields are saved. */
Field_Order GetOrder();

/** get the total number of field. */
int GetFieldNum();
/** get field at the specified index. NULL will be returned if index is out of range. */
CAttributeField* GetField(int nIndex);

/**
* get field index of a given field name. -1 will be returned if name not found.
* @param sFieldname
* @return
*/
int GetFieldIndex(const char* sFieldname);

/** return NULL, if the field does not exists */
CAttributeField* GetField(const char* sFieldname);

protected:
    int m_nClassID;
    const char* m_sClassName;
    const char* m_sClassDescription;
    vector<CAttributeField> m_attributes;
    Field_Order m_nCurrentOrder;
private:
    /** insert a new field. return true if succeeded.
     * @param bOverride: true to override existing field if any. This is usually set to true, so that inherited
     class can
     override the fields installed previously by the base class.
    */
    bool InsertField(CAttributeField& item, bool bOverride);
};

```

16.2.3 Attribute Field Interface

Class that supports attribute query needs to be derived from the IAttributeField base class.

```

/** A common interface for all classes implementing IAttributeFields
 By implementing this class's virtual functions, it enables a class to easily expose attributes
 to the NPL scripting interface. All standard attribute types are supported by the property dialog UI,

```

which makes displaying and editing object attributes an automatic process. This class has no data members, hence there are no space penalties for implementing this class. The attribute information for each class is kept centrally in a global table.

most objects in ParaEngine implement this class, such as CBaseObject, etc.

The following virtual functions must be implemented:

```
    GetAttributeClassID(), GetAttributeClassName(), InstallFields()  
*/  
class IAttributeFields  
{  
public:  
    IAttributeFields(void);  
    ~IAttributeFields(void);  
  
public:  
    ///////////////////////////////  
    // implementation of IAttributeFields  
  
    /** attribute class ID should be identical, unless one knows how overriding rules work.*/  
    virtual int GetAttributeClassID(){return ATTRIBUTE_CLASSID_IAttributeFields;}  
    /** a static string, describing the attribute class object's name */  
    virtual const char* GetAttributeClassName(){static const char name[] = "Unknown"; return name;}  
    /** a static string, describing the attribute class object */  
    virtual const char* GetAttributeClassDescription(){static const char desc[] = ""; return desc;}  
    /** this class should be implemented if one wants to add new attribute. This function is always  
    called internally.*/  
    virtual int InstallFields(CAttributeClass* pClass, bool bOverride);  
  
    ///////////////////////////////  
    //  
    // implementation of the following virtual functions are optional  
    //  
    ///////////////////////////////  
    /** whether some of the fields are modified. It is up to the implementation class to provide this  
    functionality if necessary. */  
    virtual bool IsModified(){return false;};  
    /** set whether any field has been modified. */  
    virtual void SetModified(bool bModified){};  
  
    /** validate all fields and return true if validation passed. */  
    virtual bool ValidateFields(){return true;};  
    /** get the recent validation message due to the most recent call to ValidateFields() */  
    virtual string GetValidationMessage(){return "";}  
  
    /**  
     * Reset the field to its initial or default value.  
     * @param nFieldID : field ID  
     * @return true if value is set; false if value not set.  
     */  
    virtual bool ResetField(int nFieldID){return false;};  
  
    /**  
     * Invoke an (external) editor for a given field. This is usually for NPL script field  
     * @param nFieldID : field ID  
     * @param sParameters : the parameter passed to the editor  
     * @return true if editor is invoked, false if failed or field has no editor.  
     */  
    virtual bool InvokeEditor(int nFieldID, const string& sParameters){return false;};  
public:  
    /** get the main attribute class object. */
```

```

CAttributeClass* GetAttributeClass();

    static HRESULT GetAttributeClassID_s(IAttributeFields* cls, int* p1) {*p1 = cls->GetAttributeClassID(); return S_OK;}
    static HRESULT GetAttributeClassName_s(IAttributeFields* cls, const char** p1) {*p1 = cls->GetAttributeClassName();return S_OK;}

    /**
     * Open a given file with the default registered editor in the game engine.
     * @param sFileName: file name to be opened by the default editor.
     * @param bWaitOnReturn: if false, the function returns immediately; otherwise it will wait for the
     * editor to return.
     * @return true if opened.
     */
    static bool OpenWithDefaultEditor( const char* sFilename, bool bWaitOnReturn = false);

private:
    /** initialize fields */
    void Init();
};

```

16.2.4 Attribute Class Script Wrapper

This is the script wrapper of the CAttributeClass.

```

/**
 * @ingroup ParaGlobal
 * it represents an attribute object associated with an object.
 * Call ParaObject::GetAttributeObject() or ParaObject::GetAttributeObject() to get an instance of this
object.
e.g. In NPL, one can write
local att = player:GetAttributeObject();
local bGloble = att:GetField("global", true);
local facing = att:GetField("facing", 0);
att:SetField("facing", facing+3.14);
local pos = att:GetField("position", {0,0,0});
pos[1] = pos[1]+100;pos[2] = 0;pos[3] = 10;
att:SetField("position", pos);
att:PrintObject("test.txt");
*/
struct ParaAttributeObject
{
public:
    IAttributeFields * m_pAttribute;
    CAttributeClass* m_pAttClass;
    /**
     * return true, if this object is the same as the given object.
     */
    bool equals(const ParaAttributeObject obj) const;

    bool IsValid() const;
    int GetClassID() const;
    const char* GetClassName() const;
    const char* GetClassDescription() const;

    void SetOrder(int order);
    int GetOrder();

    int GetFieldNum();
    const char* GetFieldName(int nIndex);
}

```

```

int GetFieldIndex(const char* sFieldname);
const char* GetFieldType(int nIndex);
bool IsFieldReadOnly(int nIndex);
const char* GetFieldSchematics(int nIndex);
const char* GetSchematicsType(int nIndex);
void GetSchematicsMinMax(int nIndex, float& fMin, float& fMax);
void GetField(const char* sFieldname, object& output);
void PrintObject(const char* file);
bool ResetField(int nFieldID);
bool InvokeEditor(int nFieldID, const string& sParameters);
// ... comments and some functions are ignored
};

```

16.2.5 Script Examples

One can visit our website at <http://www.paraengine.com> and download our demos; there are hundreds of script files to study under the installation directory. In the Appendix of this Book, there are also some NPL code samples to study.

16.3 Summary and Outlook

In this chapter, we have discussed and shown several useful design paradigms in the scripting system of a computer game engine. We will end this chapter by an overview of the NPL scripting system which will be discussed in detail in the second volume of the book.

16.3.1 NPL Feature Overview

Neural Parallel Language (or NPL) is an extension programming language which separates code logics from its hardware network environment. It is a new programming standard designed to support (1) frequent code updates, (2) complex graphic user interface and human-computer interactions, and (3) reconfigurable code deployment in a distributed network environment. Its extension interface and basic syntax is based on Lua, a popular light-weighted extensible language. NPL offers good support for object-oriented programming, functional programming, data-driven programming, and most importantly, network-transparent programming.

Feature 1: Network-transparent programming

“Network-transparent” means that the source code of NPL does not concern the topology of the actual networked hardware environment. NPL separates code logics from its hardware network environment. In other words, the same source code (files) can be deployed to run on one computer or a dozen of networked computers. The source files are still editable and configurable after a successful deployment, which means that one can continue apply changes to the source code or even splitting files to further deploy on more NPL runtimes.

Feature 2: Native language support for distributed visualization

Traditionally, distributed system has one or several mirrored visualization or graphic user interface (GUI) terminals. NPL proposes a new computing paradigm, which allows visualizations as well as user interactions to occur on any machine of the distributed system

concurrently. Each specialized view will be automatically computed using the same source code.

Feature 3: Distributed namespace architecture support

In NPL, script files can be logically organized in namespaces. Each NPL runtimes maintains a dynamic mapping from namespaces to physical address (IP address); hence by modifying this mapping, script files can have different physical configurations on the actual network hardware. However, name collision is inevitable for distributed systems. One reason is that there is no central boss directing all computers on the network; instead computers are either on their own or working in clusters. Another reason is that the same source code may be instanced (duplicated) thousands of times on a shared network (e.g. the Internet). NPL's solution to the namespace mapping problem is that it maintains a dynamic mapping on a task basis. In other words, a namespace mapping is only valid for some task. Here, task means job done cooperatively by a small group of computers at certain time. Hence, by imposing a few simple rules for programmers, we can eliminate problems during namespace mapping.

16.3.2 Summary and Outlook of NPL

Scripting is the symbol of flexibility and has become ubiquitous in modern computer game engines. Scripting alone means two things: (1) script files are automatically distributed and logics written in a script can be easily modified; (2) script code may be generated by dedicated visual language and software tools. In a computer game, almost all kinds of static data and most dynamic logic have text-based presentations outside the hard core of its engine. The adoption of scripting technology makes level design or game world logic composing easier than ever.

Data exchange on the Internet is also largely text-based. An entity on the Internet with or without computing capabilities automatically becomes a global resource and can be referenced by other resources. A great deal of web technologies and recommended standards have been recently proposed to make the web more and more meaningful, interactive and intelligent. As envisaged by web3d¹⁷, web service and ubiquitous computing research, etc, software applications in the future are highly distributed and cooperative. Computer games and other virtual reality applications are likely to become the most pervasive forces in pushing these web technologies into commercial uses. It is likely that one day the entire Internet would be inside one huge game world. However, two related issues must be resolved first, which are distributed computing and visualization. An extensible language system provides means and solutions to both of them. NPL is our proposal which comes with plenty of application demos and will bring more in the near future.

¹⁷ Web3D Consortium, <http://www.web3d.org/>

Chapter 17 File System

As we have shown in the early chapter 3, a modern computer game may have tens of thousands of resource files, most of which are ready-only. Deploying so many uncompressed files can be both time and space consuming. Moreover, some games may need to keep different versions of the same resource files on a user's computer, such as applying different UI themes or running the game before and after a resource patch. If you observe several 3D games' installation directories, 80% chances are that you will find one or more very large game files. It has almost become a tradition that game resource files are compressed and archived in a few large resource files at release time.

In this chapter, we will look into a typical game file system implementation. The content in this chapter is quite optional for a game engine. However, if you are at the stage of shipping demos or games, this is something worth implementing.

17.1 Foundation

During the game development, there is only one file system, which is the windows file system. Most game development team uses version control software, which only works with the windows file system. At game release time, all game resource files are in their final version; we can put them in one or several isolated virtual file systems. Each virtual file system is stored in a single archive file. Popular archive file formats are ZIP, RAR, 7-ZIP, etc. An archive file usually consists of a chunked array of compressed files and a central directory for fast indexing file addresses by their file names.

The game engine usually maintains a list of mounted file systems. The beginning of the list is the windows file system. And the rest is the virtual file systems. The game engine provides API to mount and unmount virtual file systems dynamically at runtime. In most cases, virtual file systems are all mounted at the beginning of the game. When the game engine needs to open a file, it tries to open it by iterating through all file systems in the list until either the file is found or all file systems have been examined. This allows us to specify priorities when searching a game file. For example, we can first search in the windows file system; if the file is not found there, we continue to search in other virtual file systems, etc. Multiple versions of the same file can thus have the same file path but stored in different file systems.

The most important interface of a virtual file system is to extract the uncompressed file data from the archive, given a file name. It is usually a two step procedure. In the first step, we search the file path string in the archive file's central directory for the file address (byte offset from the beginning of the archive file). In the second step, we decompress the file and copy it to a memory block and return the memory file object to the caller. In most cases, virtual file systems only store read-only files such as textures, models, sounds, and some script files. Hence, file compression, deletion and insertion, which are usually time-consuming, only occur before production time.

17.2 Architecture

In ParaEngine, there are three modules in the game file system. They are CFileManager, CArchive and CParaFile, which are shown in Figure 11.1.

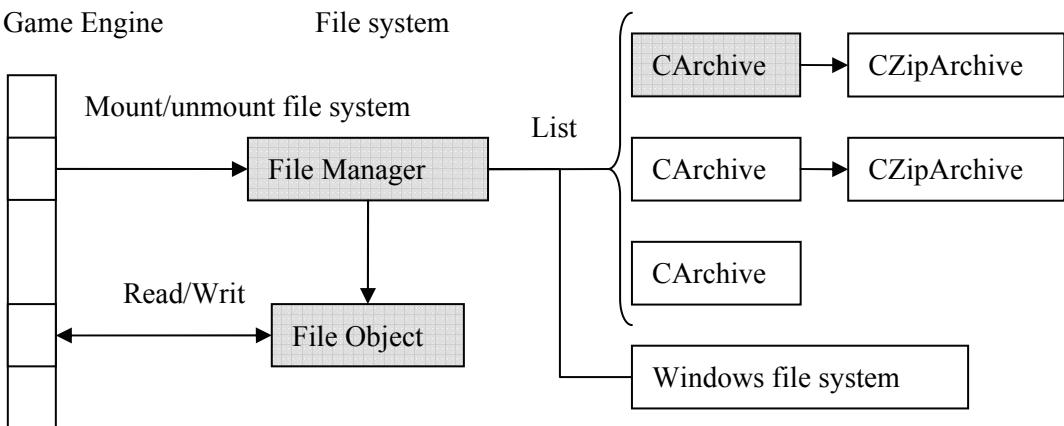


Figure 17.1 Game file system overview

CFileManager is the main file manager. One can dynamically mount and unmount virtual file systems through this class. Its main interface is shown below.

```

/*
 * This is the main file interface exposed by ParaEngine.
 * It is mainly used as a singleton class.
 */
class CFileManager
{
protected:
    list <CArchive*> m_archivers;
public:
    /** add archive to manager */
    bool OpenArchive(const string& path);
    void CloseArchive(const string& path);

    CSearchResult* SearchFiles(const string& sRootPath, const string& sFilePattern, const string&
sZipArchive, int nSubLevel=0, int nMaxFilesNum=50, int nFrom=0);

    /**
     * Check whether a given file exists on disk.
     * @param filename: file name to check
     */
    bool DoesFileExist(const char* filename);

// ...many functions are ignored
};

```

CArchive is the base class to all virtual file systems. The most important virtual file implementation is CZipArchive, which is for accesing a ZIP format archive file. Below is the CArchive interface.

```

/** file archiver base class. */
class CArchive

```

```

{
public:
    CArchive(void):m_archiveHandle(NULL), m_bOpened(false),m_nPriority(0){};
    virtual ~CArchive(void);
    inline static DWORD TypeID() {return 0;};
    virtual DWORD GetType(){ return TypeID();};

protected:
    string      m_filename;
    FileHandle m_archiveHandle;
    bool        m_bOpened;
    int         m_nPriority;

public:
    const string& GetArchiveName() const;
    /** open archive
     * @param nPriority: the smaller the number, the higher the priority with regard to other archive
     */
    virtual bool Open(const string& sArchiveName, int nPriority);

    /** close archive */
    virtual void Close(){}
    /**
     * Check whether a given file exists
     * @param filename: file name to check
     */
    virtual bool DoesFileExist(const string& filename)=0;

    /**
     * Open a file for immediate reading.
     * call getBuffer() to retrieval the data
     * @param filename: the file name to open
     * @param handle to the opened file.
     * @return : true if succeeded.
     */
    virtual bool OpenFile(const char* filename, FileHandle& handle) = 0;

    /** get file size.*/
    virtual DWORD GetFileSize(FileHandle& handle) = 0;

    /** read file.*/
    virtual bool ReadFile(FileHandle& handle,LPVOID lpBuffer,DWORD nNumberOfBytesToRead,LPDWORD lpNumberOfBytesRead) = 0;

    virtual bool WriteFile(FileHandle& handle,LPCVOID lpBuffer,DWORD nNumberOfBytesToWrite,LPDWORD lpNumberOfBytesWritten){return false;};

    /** close file.*/
    virtual bool CloseFile(FileHandle& hFile) = 0;

    /** create a new file for writing
     * @param bAutoMakeFilePath: if true, the file path will be created, if not exists
     */
    virtual bool CreateNewFile(const char* filename, FileHandle& handle, bool bAutoMakeFilePath = true){return false;};
}

```

The file object or CParaFile is an all-in-one class for file access, which is used at other places of the game engine. A CParaFile object may represent a disk file, a resource file, a memory file read from a virtual file system, or a newly created disk file, etc. It provides open

and close file functions, read and write file functions, and a great deal of other helper functions for extracting binary or formatted data from the file.

17.3 Code

The most difficult implementation code of a game file system is directory or index building and file decompression. Both of them are part of the CArchive class implementation. We will only take a look at the most commonly used virtual file archive format, the ZIP format. For zip files, there is a widely used single file compression and decompression library called ZLib, which can be downloaded for free from the web. We guess that over 90% games use this lib for file decompression. Please read the ZLib header file for its usage, basically it is just one function for compression and the other for decompression.

However, before we can decompress a file, we must first locate the compressed file inside the ZIP file archive. Zip file is a chunked file format, whose specification can be found on the web. Figure 17.2 shows the overview of a Zip file format.

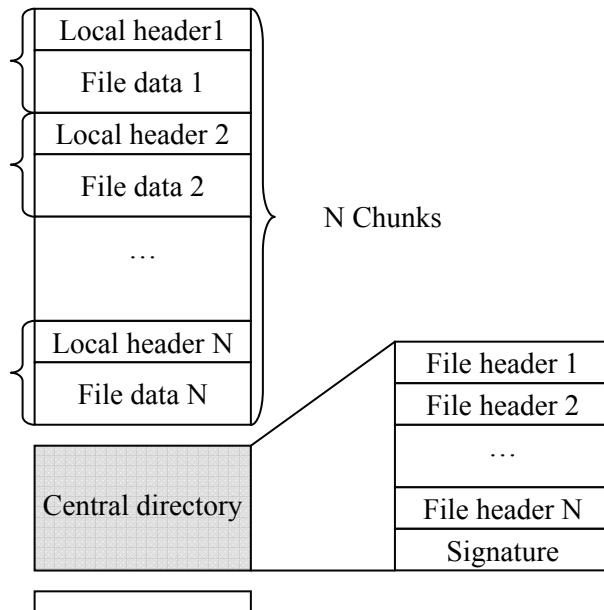


Figure 17.2 Zip archive format overview

It is common for a zip file to contain over 10,000 files with a total size of several hundred MB. It is not efficient to traverse the file from the beginning to build the file directory. Instead we will seek to the end of zip file, locate the central directory by its signature and build the file directory from there.

Since we are not dealing with a super large file system, we can read the entire directory to memory at once and save them in an array sorted by file path names. It takes about 0.1 second to read and build file directory for a ZIP archive containing 20,000 files on a typical 3 G Hz PC. So it does not take long to load the directories of several large ZIP archives to memory at the game loading time.

The following are code snippet to build file directory from a ZIP archive.

```

bool CZipArchive::ReadEntries()
{
    char tmp[1024];
    long locatedCentralDirOffset = LocateBlockWithSignature(ZIP_CONST_ENDSIG, m_pFile->getSize(), sizeof(ZIP_EndOfCentralDirectory), 0x4);
    if (locatedCentralDirOffset < 0) {
        long locatedCentralDirOffset = LocateBlockWithSignature(ZIP_CONST_ENDSIG, m_pFile->getSize(), sizeof(ZIP_EndOfCentralDirectory), 0xffff);
        if (locatedCentralDirOffset < 0)
            return false;
    }

    ZIP_EndOfCentralDirectory EndOfCentralDir;
    m_pFile->read(&EndOfCentralDir, sizeof(ZIP_EndOfCentralDirectory));

    if(EndOfCentralDir.commentSize>0)
    {
        SAFE_DELETE_ARRAY(m_zipComment);
        m_zipComment = new char[EndOfCentralDir.commentSize];
        m_pFile->read(m_zipComment, EndOfCentralDir.commentSize);
    }
    int offsetOfFirstEntry = 0;
    if (EndOfCentralDir.offsetOfCentralDir < locatedCentralDirOffset - (4 + EndOfCentralDir.centralDirSize)) {
        offsetOfFirstEntry = locatedCentralDirOffset - (4 + EndOfCentralDir.centralDirSize + EndOfCentralDir.offsetOfCentralDir);
        if (offsetOfFirstEntry <= 0) {
            return false;//throw new ZipException("Invalid SFX file");
        }
    }
    m_pFile->seek(offsetOfFirstEntry + EndOfCentralDir.offsetOfCentralDir, false);

    IReadFile* pReader = NULL;

#ifdef ENABLE_INDEX_CACHE
    CMemReadFile memfile(m_pFile, locatedCentralDirOffset - m_pFile->getPos());
    pReader = &memfile;
#else
    pReader = m_pFile;
#endif

    int nEntryNum = EndOfCentralDir.entriesForThisDisk;
    m_FileList.set_used(nEntryNum);
    m_FileList.set_sorted(false);
    SAFE_DELETE_ARRAY(m_pEntries);
    m_pEntries = new SZipFileEntry[nEntryNum];
    for (int i = 0; i < nEntryNum; ++i)
    {
        ZIP_CentralDirectory CentralDir;
        pReader->read(&CentralDir, sizeof(ZIP_CentralDirectory));
        if(CentralDir.Sig!=ZIP_CONST_CENSIG)
        {
            return false; // throw new ZipException("Wrong Central Directory signature");
        }

        m_FileList[i].m_pEntry = &m_pEntries[i];
        if(m_FileList[i].m_pEntry == 0)
            return false;
    }
}

```

```

SZipFileEntry& entry = *(m_FileList[i].m_pEntry);

// read filename
entry.zipFileName.reserve(CentralDir.NameSize+2);
pReader->read(tmp, CentralDir.NameSize);
tmp[CentralDir.NameSize] = 0x0;
entry.zipFileName = tmp;
if (m_bIgnoreCase)
    make_lower(entry.zipFileName);

entry.CompressionMethod = CentralDir.CompressionMethod;
entry.UncompressedSize = CentralDir.UnPackSize;
entry.CompressedSize = CentralDir.PackSize;

if (CentralDir.ExtraSize > 0) {
    pReader->seek(CentralDir.ExtraSize, true);
}

if (CentralDir.CommentSize > 0) {
    pReader->seek(CentralDir.CommentSize, true);
}
// calculate data pos offset.
int nDataPos = CentralDir.LocalHeaderOffset + sizeof(SZIPFileHeader) + CentralDir.NameSize;

entry.fileDataPosition = nDataPos;
}
return true;
}

```

17.4 Summary and Outlook

Many game engine modules need to access files. It is good practice to provide a unified file access interface throughout the game engine. In this chapter, we have illustrated a typical file system in a game engine.

A number of functionalities can be realized by extending the game file system, such as XML data serialization, file patch system, HTTP and FTP file download, etc.

Chapter 18 Frame Rate Control

Computer game engine is a real-time visualization and simulation application. However, the computation complexity of each frame in a computer game engine is not steady. Time management (including time synchronization and frame rate control) is the backbone system that feedbacks on a number of game engine modules to provide physically correct, interactive, stable and consistent graphics output.

In this chapter, we will look at timing issues in a game engine. This chapter could be useful to developers who have already built a fairly complete game engine. This is why we put it at the end of the book.

18.1 Foundation

Timing or frame rate in game engines is both unpredictable and intertwined for a number of reasons. For example: resource files are dynamically loaded from the file system; some scene entities are animated independently, whereas others may form master-slave animation relationships; simulation and graphics routines are running at unstable (frame) rate; some of the game scene entities are updated at variable-length intervals from multiple network servers; and some need to swap between several LOD (level-of-detail) configurations to average the amount of computations in a single time step. In spite of all these things, a game engine must be able to produce a stable rendering frame rate, which best conveys the game state changes to the user. For example, a physically correct animation under low frame rate (loosing many rendering frames in the middle) is sometimes less satisfactory than time-scaled animation, where continuities in character motions are preferred. A solution to the problem is to use statistical or predictive measures to calculate the length of the next time step for different time-driven game processes and objects. In other words, time management architecture (such as the one proposed in this chapter) is worth to be integrated to a computer game engine.

It is also important to realize that timing in computer games is different from that of the reality and other simulation systems. A computer game prefers (1) interactivity or real-time game object manipulation (2) consistency (e.g. consistent game states for different clients) (3) stable frame rate (this is different from interactive or average frame rate). For technical reasons, it is unrealistic to synchronize all clocks in the networked gaming environment to one universal time. Even if we are dealing with one game world on a standalone computer, it is still not possible to achieve both smoothness and consistency for all time related events in the game. Fortunately, by rearranging frame time, we can still satisfy the above game play preferences, while making good compromises with the less important ones, such as physical correctness.

18.1.1 Overview

Time management has been studied in a number of places of a computer game engine, such as (1) variable frame length media encoding and transmission (2) time synchronization for distributed simulation (3) game state transmission in game servers (4) LOD based interactive frame rate control for complex 3D environments. However, there have been relatively few literatures on a general architecture for time synchronization and frame rate control, which is immediately applicable to an actual computer game engine. In a typical game engine,

modules that need time management support include: rendering engine, animation controller, I/O, camera controller, physics engine, network engine (handling time delay and misordering from multiple servers), AI (path-finding, etc), script system, in-game video capturing system and various dynamic scene optimization processes such as ROAM based terrain generation, shadow generation and other LOD based scene entities. In addition, in order to achieve physically correct and smooth game play, frame rate of these modules must be synchronized, and in some cases, rearranged according to some predefined constraints.

In the game development forums, many questions have been asked concerning jerky frame rates, jumpy characters and inaccurate physics. In fact, these phenomena are caused by a number of coordinating modules in the game engine and cannot be easily solved by a simple modification.

In the following sections, we will identify frame rate related problems in a game engine.

18.1.2 Decoupling Graphics and Computation

Several visualization environments have been developed which synchronize their computation and display cycles. These virtual reality systems separate the graphics and computation processes, usually by distributing their functions among several platforms or system threads (multi-threading).

When the computation and graphics are decoupled in an unsteady visualization environment, new complications arise. These involve making sure that simultaneous phenomena in the simulation are displayed as simultaneous phenomena in the graphics, and ensuring that the time flow of computation process is correctly reflected in the time flow of the displayed visualizations (although this may not need to be strictly followed in some game context). All of this should happen without introducing delays into the system, e.g. without causing the graphic process to wait for the computation to complete. The situation is further complicated if the system allows the user to slow, accelerate or reverse the apparent flow of time (without slowing or stopping the graphics process), while still allowing direct manipulation and exploration at real time.

However, decoupling graphics and computation is a way to explore parallelism in computing resources (e.g. CPUs and GPUs), but it is not a final solution to time management problems in game engines. In fact, in some cases, it could make the situation worse; not only because it has to deal with complex issues such as thread-safety (data synchronization), but also because we may lose precise control over the execution processes. E.g. we have to rely completely on the operating system to allocate time stamps to processes. Time stamp management scheme supported by current operating system is limited to only a few models (such as associating some priority values to running processes). In game engines, however, we need to create more complex time dependencies between processes. Moreover, data may originate from and feed to processes running on different places via unpredictable media (i.e. the Internet). Hence, our proposed architecture does not rely on software or hardware parallelism to solve the frame rate problem.

18.1.3 I/O

The timing module for IO mainly deals with when and how often the engine should process user commands from input devices. These may include text input, button clicking, camera

control and scene object manipulation, etc. Text input should be real-time; button clicking should subject to rendering rate. The tricky part is usually camera control and object manipulation. Unsmooth camera movement in 3D games will greatly undermine the gaming experience, especially when camera is snapped to the height and norm of the terrain below it. Direct manipulation techniques allow players to move a scene object to a desired location and view that visualization after a short delay. While the delay between a user control motion and the display of a resulting visualization is best kept less than 0.2 seconds, experience has shown that delays in the display of the visualization of up to 0.5 seconds for the visualization are tolerable in a direct manipulation context. Our experiment shows that camera module reaction rate is best set to constant (i.e. independent of other frame rates).

18.1.4 Frame Rate and Level of Detail

The largest number of related work lies in Frame rate and Level of detail (LOD). The idea of LOD techniques is that we can maintain stable frame rate by maintaining a stable scene complexity or the number of rendered triangles per frame.

In many situations, a frame rate, lower than 30 fps, is also acceptable by users as long as it is stable. However, a sudden drop in frame rate is rather annoying since it distracts the user from the task being performed. To achieve constant frame rate, scene complexity must be adjusted appropriately for each frame. In indoor games (where the level geometry may be organized by BSP nodes or PVS), the camera is usually inside a closed room. Hence, the average scene complexity can be controlled fairly easily by level designers. The uncontrolled part is mobile characters, which are usually rendered in relatively high poly models in modern games. Yet, scene complexity can still be controlled by limiting the number of high-poly characters in their movable region, so that the worst case polygon counts of any screen shot can stay below a predefined value. In multiplayer Internet games, however, most character activities are performed in outdoor scenery which is often broader (having much longer line-of-sight) than indoor games. Moreover, most game characters are human avatars. It is likely that player may, now and then, pass through places where the computer cannot afford to sustain a constant real-time frame rate (e.g. 30 FPS). The proposed frame rate controller architecture can ease such situations, by producing smooth animations even under low rendering frame rate.

18.1.5 Network servers

Another well study area concerning time management is distributed game servers. In peer-to-peer architecture or distributed client/server architecture, each node may be a message sender or broadcaster and each may receive messages from other nodes simultaneously.

In order for each node to have a consistent or fairly consistent view of the game state, there needs to be some mechanism to guarantee some global ordering of events. This can either be done by preventing misorderings outright (by waiting for all possible commands to arrive), or by having mechanisms in place to detect and correct misorderings. Even if visualization commands from the network can be ordered, game state updates on the receiving client still needs to be refined in terms of frame rate (time step) for smooth visualization. Another complication is that if a client is receiving commands from multiple servers, the time at which one command is executed in relation to others may lead to further ordering constraints.

The ordering problem for a single logical game server can usually be handled by designing new network protocols which inherently detects and corrects misorderings. However, flexible time control cannot be achieved solely through network protocols. For example, in case several logical clocks are used to totally order events from multiple game servers, the game engine must be able to synchronize these clocks and use them to compose a synthetic game scene. Moreover, clocks in game engines are not directly synchronized. For example, some clocks may tick faster, and some may rewind. Hence, time management in game development can be very chaotic if without proper management architecture.

18.1.6 Physics Engine

The last category of related works that will be discussed is timing in physics engine, which is also the trickiest part of all.

The current time in the physics engine is usually called simulation time. Each frame, we advance simulation time in one or several steps until it reaches the current rendering frame time (However, we will explain later that this is not always necessary for character animation under low frame rate). Choosing when in the game loop to advance simulation and by how much can greatly affect rendering parallelism. However, simulation time is not completely dependent on rendering frame time. In case the simulation is not processing fast enough to catch up with the rendering time, we may need to freeze the rendering time and bring the simulation time up to the current frame time, and then unfreeze. Hence it is a bi-directional time dependency between these two time-driven systems.

Integrating Key Framed Motion

In game development, most game characters, complicated machines and some moving platforms may be hand-animated by talented artists. Unfortunately, hand animation is not obligated to obey the laws of physics and they have their own predefined reference of time.

To synchronize the clocks in the physics engine, the rendering engine and the hundreds of hand-animated mesh objects, we need time management framework and some nonnegotiable rules. For example, we consider key framed motion to be nonnegotiable. A key framed sliding wall can push a character, but a character cannot push a key framed wall. Key framed objects participate only partially in the simulation; they are not moved by gravity, and other objects hitting them do not impart forces. They are moved only by key frame data. For this reason, the physics engine usually provides a callback function mechanism for key framed objects to update their physical properties at each simulation step. Call back function is a C++ solution to this paired action (i.e. the caller function has the same frame rate as the call back function). Yet, calculating physical parameters could be computationally expensive. E.g. in a skeletal animation system, if we want to get the position of one of its bones at a certain simulation time, we need to recalculate all the transforms from this bone to its root bone.

18.2 Architecture

In this section, we propose the Frame Rate Control (FRC) architecture and show how it can be integrated in a game engine.

18.2.1 Definition of Frame Rate and Problem Formulation

In the narrow sense, frame rate in computer graphics means the number of images rendered per second. However, the definition of frame rate used in this paper has a broader meaning. We define frame rate to be the activation rate of any game process. More formally, we define $f(t) \rightarrow \{0,1\}$, where t is the time variable and $f(t)$ is the frame time function. We associate a process in the game engine to a certain $f(t)$ by the following rule: $f(t)$ is 1, if and only if its associated process is being executed. The frame rate at time t is defined to be the number of times that the sign of $f(t)$ changes from 0 to 1, during the interval $(t-1,t]$.

Let $\{t^{s_k} | k \in N, \lim_{\delta x \rightarrow 0} (\frac{f(t^k) - f(t^k - \delta x)}{\delta x}) = +\infty\}$ be a set of points on t , where the value of $f(t)$ changes from 0 to 1. Let $\{t^{e_k} | k \in N, \lim_{\delta x \rightarrow 0} (\frac{f(t^k + \delta x) - f(t^k)}{\delta x}) = -\infty\}$ be a set of points on t , where the value of $f(t)$ changes from 1 to 0. Also we enforce that $\forall k (s_k < e_k < s_{k+1})$. $\{t^{e_k}, t^{s_k}\}$ is equivalent to $f(t)$ for describing frame time function.

With the above formulation, we can analyze and express the frame rate of a single function as well as the relations between multiple frame rate functions easily. Figure 18.1 shows the curves of three related frame rate functions: i, j and k.

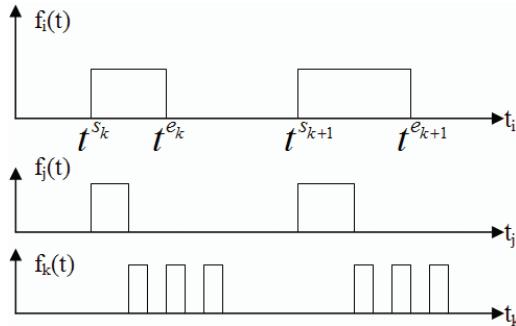


Figure 18.1 Sample curves of frame rate functions

The curve of $f(t)$ may be unpredictable in the following ways.

- In some cases, the length of time when $f(t)$ remains 1 is unpredictable (i.e. $|t^{e_k} - t^{s_k}|$ is unknown), but we are able to control when and how often the $f(t)$ changes from 0 to 1 (i.e. t^{s_k} can be controlled). The rendering frame rate is often of this type.
- In other cases, we do not know when the value of $f(t)$ will change from 0 to 1 (i.e. t^{s_k} is unknown), but we have some knowledge about when $f(t)$ will become 0 again (i.e. we know something about $|t^{e_k} - t^{s_k}|$). The network update rate is often of this type.
- In the best cases, we know something about $|t^{e_k} - t^{s_k}|$ and we can control t^{s_k} . The physics simulation rate is often of this type.
- In the worst cases, only statistical knowledge or a recent history is known about $f(t)$. The video compression rate for real-time game movie recording and I/O event rate are often of this type (fortunately, they are also easy to deal with, since these frame rates are independent and do not need much synchronization with other modules.).

Let $\{f_n(t_n)\}$ be a set of frame time functions, which represent the frame time for different modules and objects in the game engine and game scene. The characteristics of $f(t)$ and the

relationships between two curves $f_i(t)$ and $f_j(t)$ can be expressed in terms of constraints. Some simple and common constraints are given below, with their typical use cases. (More advanced constraints may be created.)

1. $|t_i - t_j| < \text{MaxDiffTime}$, ($\text{MaxDiffTime} \geq 0$) . Two clocks i, j should not differentiate too much or must be strictly synchronized. The rendering frame rate and physics simulation rate may subject to this constraint.
2. $(t_i - t_j) < \text{MaxFollowTime}$, ($\text{MaxFollowTime} > 0$) . Clock(j) should follow another clock(i). The rendering and IO (user control such as camera movement) frame rate may subject to it.
3. $\forall_{k,l}((t_i^{s_k}, t_i^{s_{k+1}}) \cap (t_j^{s_l}, t_j^{s_{l+1}}) = \emptyset)$. Two processes i, j cannot be executed asynchronously. Most local clocks are subject to this constraint. If we use single-threaded programming, this will be automatically guaranteed.
4. $\max\{(t^{s_k} - t^{s_{k-1}})\} < \text{MaxLagTime}$. The worst cast frame rate should be higher than $1/\text{MaxLagTime}$. Physics simulation rate is subject to it for precise collision detection.
5. $\forall k(|t^{s_{k+1}} + t^{s_{k-1}} - 2t^{s_k}| < \text{MaxFirstOrderSpeed})$. There should be no abrupt changes in time steps between two consecutive frames. The rendering frame rate must subject to it or some other interpolation functions for smooth animation display.
6. $\forall k((t^{s_k} - t^{s_{k-1}}) = \text{ConstIdealStep})$. Surprisingly, this constraint has been used most widely. Games running on specific hardware platform or with relatively steady scene complexity can use this constraint. Typical value for ConstIdealStep is 1/30fps, which assumes that the user's computer must finish computing within this interval. In in-game video recording mode, almost all game clocks are set to this constraint.
7. $\forall k((t^{s_k} - t^{s_{k-1}}) \leq \text{ConstIdealStep})$. Some games prefer setting their rendering frame rate to this constraint, so that faster computers may render at a higher rate. Typical value for ConstIdealStep is 1/30fps; while at real time $(t^{s_k} - t^{s_{k-1}})$ may be the monitor's refresh rate.

18.2.2 Integrating Frame Rate Control to the Game Engine

There can be many ways to integrate frame rate control mechanism in a game engine and it is up to the engine designer's preferences. We will propose here the current integration implementation in ParaEngine.

In ParaEngine, we designed an interface class called FRC Controller and a set of its implementation classes, each of which is capable to manage a clock supporting some timing constraints. Instances of FRC Controller are created and managed in a global place (such as in a singleton class). A set of global functions (see Time Scheme Manager in Figure 18.2) are used to set the current frame rate management scheme in the game engine. Each function will configure the frame rate controller instances to some specific mode. For example, one such function may set all the FRC controllers for video capturing at a certain FPS; another function may set the FRC controllers so that game is paused but 2D GUI is active; yet a third function may set controllers so that the game is running normally with an ideal 30 FPS frame rate.

Like in most computer game engines, a scene manager is used for efficient game object management. For each time-driven process and object in the scene, the game engine must know which FRC controllers it is associated with. One way to do it is to keep a handle or

reference to the FRC controllers in every scene object. However, it is inefficient in terms of management and memory usage. A more efficient way to do this is to take advantage of the tree hierarchy in the scene manager and its transversal routines during rendering and simulation. This is done by creating a new type of dummy scene node called time node (see Figure 18.2), which contains the reference or handle to one or several FRC controller instances. Then they are inserted to the scene graph like any other scene nodes. Finally, the following rules are used to retrieve the appropriate FRC controllers for a given scene object:

- The FRC controllers in a time node will be applied to all its child scene nodes recursively.
- If there is any conflict among FRC controllers for the current scene node, settings in the nearest time node in the scene graph are adopted.

Since frame rate controllers are managed as top-layer (global) objects in the engine (see Figure 18.2). Any changes made to the FRC controllers will be immediately reflected in the next scene traversal cycle. Three global frame rate controllers are used: IO_TIMER, SIM_TIMER and RENDER_TIMER. These are the initial FRC controllers passed to processes in the game loop. As a process goes through the scene graph, the initial settings will be combined with or overridden by FRC controller settings contained in the time nodes.

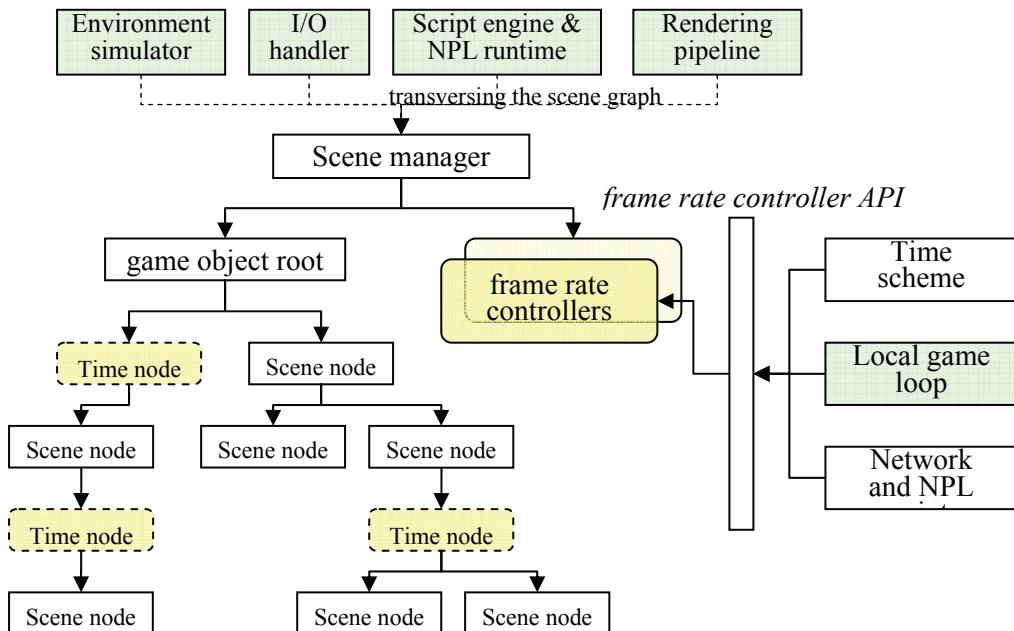


Figure 18.2 Integrating time control to the game engine

18.3 Evaluation

This section contains some use cases of the proposed framework. The combination of FRC controller settings can create many interesting time synchronization schemes, yet we are able to demonstrate just a few of them here.

18.3.1 Frame Rate Control in Video Capturing

The video system in ParaEngine can create an AVI video while the user is playing the game. When high-resolution video capture mode (with codec) is on, the rendering frame rate may drop to well below 5 FPS. It's a huge impact, but fortunately it does not get run at production time. The number of frames it will produce depends on the video FPS settings, not the game FPS when the game is being recorded.

Now a problem arises: how do we get a 25 FPS output video clip, while playing the game at 5FPS? In such cases, the time management scheme should be changed for the following modules: I/O, physics simulation, AI scripting and graphics rendering. Even though the game is running at very low frame rate, it should still be interactive to the user, generate script events, perform accurate collision detection, run environment simulation and play coordinated animations, etc, as if the game world is running precisely at 25 FPS. ParaEngine solves this problem by swapping between two sets of FRC controller schemes for clocks used by its engine modules. In normal game play mode, N-scheme is used; whereas during video capturing, C-scheme is used. See Table 18.1. In C-scheme, the simulation and the scripting system use the same constant-step frame rate controller as the rendering and I/O modules. The resulting output of C-scheme is that everything in the game world is slowed but still interactive.

Table 18.1 Frame rate control schemes

	N-scheme	C-scheme
<i>ConstIdealFPS</i>	30 or 60	20 or 25
Rendering	FRC_CONSTANT	FRC_CONSTANT
I/O	FRC_CONSTANT	FRC_CONSTANT
Sim & scripting	FRC_FIRSTORDER	FRC_CONSTANT

18.3.2 Coordinating Character Animations

In computer game engine, a character's animation is usually determined by the combination of its global animation and local animation. Global animation determines the position and orientation of the character in the scene, which is usually obtained from the simulation engine. The local animation usually comes from pre-recorded animation clips. In order for the combined motion of the character to be physically correct, the simulation time is usually strictly matched with the local animation time using constraint (1) in Section 3.1. However, this is not the best choice for biped animation with worst case rendering frame rate between 10FPS and 30 FPS. Our experiment shows that setting simulation time to constraint (5) and the local animation time to constraint (6) will produce more satisfactory result. This configuration does not generate strictly correct motion, but it does produce smooth and convincing animation. The explanation is given below. Suppose a biped character is walking from point A to B at a given speed. Assume that the local "walk" animation of the biped takes 10 frames at its original speed (i.e. it loops every 10 frames). Suppose that the simulation engine needs to advance 20 frames in order to move the biped from A to B at the biped's original speed. Now consider two situations. In situation (i), 20 frames can be rendered between A and B. In situation (ii), only 10 frames can be rendered. With constraint (1), the

biped will move fairly smoothly under situation (i), but appears very jerky under situation (ii). This is because if the simulation and the local animation frame rates are strictly synchronized, the local animation might display frame 0, 2, 4, 6, 8, 1, 3, 5, 7, 9 at its best; in the actual case, it could be 0, 1,2, 8,9, 1,2,3,7,9, both of which are missing half the frames and appears intolerable jumpy. However, with constraint (5) and (6) applied, the local animation frame displayed in situation (i) will be 0,1,2,3,4,5,6,7,8,9, 0,1,2,3,4,5,6,7,8,9, and under situation (ii), 0,1,2,3,4,5,6,7,8,9, both of which play the intact local animation and look very smooth. The difference is that the biped will stride a bigger step in situation (ii). But experiment shows that users tend to misperceive it as correct but slowed animation. The same scheme can be used for coordinating biped animations in distributed game world. For example, if there is any lag in a biped's position update from the network, the stride of the biped will be automatically increased, instead of playing a physically correct but jumpy animation.

18.4 Summary and Outlook

The main purpose of this chapter is to help readers identifying frame rate related issues in a game engine. We also propose the frame rate architecture in ParaEngine.

Ad-hoc timing control, as frequently seen in released product, needs to be refined by their designers. In the long run, time management will become very important in distributed visualization and simulation of virtual environment, such as networked computer game worlds.

Chapter 19 Summary and Outlook of Game Engine

After reading this book, we may get an overwhelming feeling about computer game engine: a computer game engine is a complicated engineering of virtually all fields of computer science in the most efficient ways. Applications created by game engine deliver the best multimedia content that a computer could possibly offer.

19.1 Age of Middleware

Although, a computer game engine embraces a large number of field technologies, they are not immediately applicable. In the old days, a game engine developer needs to recode a great deal of them in order to meet the performance requirement of game engine. Nowdays, common functional modules of a game engine are being specialized and commericialized. They can be small to medium modules, such as game physics, particle simulation, AI modules, network modules, scripting system, terrain engine, vegetation rendering; or they can be major components of game engine, such as the entire graphics system, world editing environment, client/server system, etc. These pieces of dedicated software are called middlewares. Sometimes, it is hard to distinguish them from the game engine itself, or the game engine itself may sometimes become a middleware.

The extensive use of middleware also requires extensive knowledge of a game engine. Successful middleware users are usually those who have already implemented their own game engine, but want to shorten the game production time and increase product quality.

On the other hand, innovative games usually require specialized game engines. No game engine today can accommodate all types of games. When the cost of customizing an existing game engine is too high, project manager needs to make the bold decision to rework major game engine modules to suit the need of the game.

19.2 World Editing

There are three types of world editing choices for game engine. They are described in the following sub sections.

19.2.1 In-game Editing

Most game engines provide some sort of in-game world editing application, which allows designers to lay-out level geometry, triggers and a variety of game objects dynamically. This is usually the case for games featuring extremely large territories, such as MMORPG games. ParaEngine supports this editing method natively.

19.2.2 Customizing over Professional 3D Authoring Tools

Optionally, a game engine may customize professional editing applications, such as 3dsmax and maya, and make it an integrated world editing environment. This is usually the case for art intensive games such as standalone console RPG games. By the time of writing, ParaEngine only supports this type of world editing via 3dsmax.

19.2.3 Writing Special Tools for Game

Most game projects do not implement their own game engine. Instead, they usually have a dedicated tool team which builds the world editing tools to the need of their specific games. One of the most demanding game types that require much work on game editor is perhaps the Real-Time Strategy (RTS) games. These specified world editing tools are usually made via API exposed by the game engine. Some game engine allows source code level access to the game engine; however, this is not the advised way to build world editing tools. Instead world editing tools are usually built via the game engine's C/C++ API or API exposed through the game engine's scripting interface. In ParaEngine, 90% of the major game engine functionalities, including the network and database logics, are exposed via the NPL scripting interface. This allows tool developers to use the scripting language to construct advanced world editing tools for their games.

19.2.4 Outlook

Unlike other super sophisticated computer software, we think that the near future of game editing environment will not be ruled by a few giants. Instead, game editing will continue to diversify for different people with different skills and objectives.

The logics of games are endless, and so does the human computer interface of its editing environment.

19.3 Next Generation Game Engine

In the past and today, graphics is the most-talked-about feature for next generation game engine. We will continue to see more realistic scenes powered by tomorrow's technology. We also believe that networking and artificial intelligence will play more important roles in a game engine. They are also the two aspects that ParaEngine aims to specialize in.

We believe that computer game is the driving force which could bring the Internet to Web 3D¹⁸.

¹⁸ Web3D Consortium, www.web3d.org

Appendix A: Code Optimization

A game engine programmer always keeps one thing in its mind which is code optimization. There are high-level algorithms and code patterns which we covered in this book. However, when writing pieces of critical programming code, there are some basic techniques a programmer should know. Since there are already many good books on the topic, we will cover just some of them which will immediately boost your unoptimized code. They are freely listed below.

Inline Functions

Recognize frequently called functions that have small bodies; if you feel that inlining might boost performance, just do it. The most typical example of using inlining is on the math or physics functions. If you observe the DirectX extension library, you will find that most vector and matrix functions are inlined.

Use Globals

Here, globals do not just mean global parameters, but also any singleton object in the game engine. A game engine usually contains many singleton objects, such as scene manager, scene states, render device, current camera, current player, etc. Although it may be bad practice elsewhere, game engines almost never pass globals as parameters to functions. The main reason is on performance.

Writing Simpler Code

Do not pack many things in to a single line of C/C++ code. The compiler may fail to optimize it due to it. Modern compiler can automatically optimize our source code by applying a number tricks to the source code, such as inlining, changing code order or duplicating code to prevent frequent branching, etc.

Branching

Currently CPU runs lots of instructions in parallel. A branch instruction such as “if” will undermine parallelism in the CPU, hence we should avoid the number of branch instructions in critical program code, and if it is truly unavoidable, we should try to increase the chance that a branch instruction passes. As a general rule, try to avoid using branching inside a loop, and put unlikely-to-execute code in the “else” part instead of “if” part.

Precompute

We can use fast sine and cosine functions which precompute values and retrieve them by indexing in to the table. Similarly, we can minimize the number of division calls by precompute its inverse.

Align data members

When space is critical, some people tend to pack data members in to bytes or even bits. However, the computer accesses data most efficiently with the default 4 bytes alignment. In most cases, system memory is not quite a scarce resource as it appears. Comparing with the

size of model and texture files in memory; saving a few bytes in data structuring does not outweigh its costs of decreased member access speed.

Profiling

There are many good commercial profiling tools. DirectX SDK also comes with a GPU profiler called PIX. When developing ParaEngine, we wrote a handy code level profiler ourselves. A game engine is mainly a looped program where many modules are executed on a regular basis. Hence we usually use the code profiler to generate information such as min, max, average execution time, execution time deviations, etc. The following shows a segment of information generated by the code profiler in ParaEngine.

```
<RebuildSceneState>
Avg: 0.000128  Dev: 0.000000  Max: 0.001101  Min: 0.000127  Total Frames: 615

<Render_Mesh>
Avg: 0.000114  Dev: 0.000555  Max: 0.062597  Min: 0.000011  Total Frames: 615

<Script FrameMove>
Avg: 0.008639  Dev: 0.038664  Max: 3.039571  Min: 0.000015  Total Frames: 669

<3D Scene Render>
Avg: 0.006790  Dev: 0.007811  Max: 0.613430  Min: 0.004168  Total Frames: 615

<Animate FrameMove>
Avg: 0.000168  Dev: 0.000254  Max: 0.065232  Min: 0.000005  Total Frames: 669

<present>
Avg: 0.000561  Dev: 0.000132  Max: 0.024704  Min: 0.000179  Total Frames: 669

<terrain_tessellation>
Avg: 0.007236  Dev: 0.001256  Max: 0.010994  Min: 0.004663  Total Frames: 56
```

Appendix B: Da Vinci World

The first object of the painter is to make a flat plane appear as a body in relief and projecting from that plane.

-- Leonardo da Vinci (1452-1519)

Leonardo da Vinci was, and still is, known as one of the greatest inventors and thinkers of the Italian Renaissance. He was an architect, musician, scientist, mathematician, and "inventor of genius." He especially excelled in science and art, and was always thinking outside of the box. Many of his ideas have been an inspiration for some modern technology.

In this special chapter, we use version 1.0 of ParaEngine SDK (personal edition) to create a sample game scene, called Da Vinci World. The complete series of "Da Vinci World" is a collection of games which allow children to experiment with and fulfill their imaginations in 3D worlds. We have tested the game among children aged 7 to 12 with some former gaming experience, and the result is quite positive. Children are more creative and experimental than we imagined. After watching the children creating game worlds for hours, adults usually reach the conclusion that children are the greatest inventors and thinkers of our time.

The explicit purpose of this chapter is to teach casual readers how to use ParaEngine to quickly build a game world. Besides that, we hope:

- To provide a tangible goal towards which new engine programmers may schedule to work.
- To allow users (programmers, artists, designers, etc) to look up the related technologies in this book, when they are using them.
- To give a simple outline of how a casual game can be developed.

In the following sections, we follow steps to create the *Da Vinci World* sample game scene. Sections whose names start with "Do It Yourself" are advanced and usually require programming via the NPL scripting interface. Casual readers can skip these sections completely.

B.1 Game Proposal and Design

Goal

Design a sample game scene, in which children can do experiments and fulfill their imaginations.

Where does the game happen?

It happens in a fantasy world surrounded by mountains with water and island in the middle. The world is beautiful but looks a bit barren at first. There should be some open flat land where buildings can be built and plants can be grown.

What do I control?

The children control one character at a time in a third person or first person perspective.

How many characters can I control?

The children can populate the world with many characters. They can toggle between characters.

What is the key point of the game?

The key is to entice the children to fulfill their imaginations and stories by creating new elements, as well as, to discovery interactive parts of the world.

How about Models, Characters, Game World Logics, UI, Effects, Audio, etc?

We will use the default ones from ParaEngine SDK demo. ParaEngine Dev Studio plan to share resources with you under a very user-friendly license. Please read the SDK for details.

Can I preview the game?

Please go to our website www.paraengine.com to download. One can quickly go through the following pages to see some screen shots.

B.2 Prepare Your Work Environment

The only requirement to make this demo game is the installation of ParaEngine SDK personal edition.

However, for more serious game making, lots of third party tools are used. For artists, you may need software like Photoshop, 3dsmax, Maya, Deep paint, etc. For designers and programmers, you need a word editor and a program editor, such as Ultraedit, or even visual studio. For all people, version control your artifacts can be top most important and some game studios choose to use version control software with user friendly interface, such as AlienBrain, Visual Source Safe, etc.

B.3 Getting Used to ParaEngine SDK

For casual users, ParaEngine SDK is just a single executable, called the *default interface*. One can create, edit and release using the same executable. It features more than a what-you-see-is-what-you-get interface. Our design principle is to make it 100 percent functional, which means that with each click of the mouse, you bring something explicit to happen in the world. In fact, it is so designed that children aged 10-12 can quickly master it themselves; any operation can do no harm to the system as long as script files are not touched from an external editor. Hence, in the rest of the book, we will not teach instructions if it is on the *default interface*.

For intermediary users, we offer another user interface, called ParaIDE, which runs side by side with the default interface. One can bring up ParaIDE by pressing F5 or select from the menu in the default interface. See Figure 19.1. The application on top is ParaIDE. The 3D application on the background is ParaEngine SDK's default interface. These two applications can communicate at runtime. The ParaIDE provides advanced functionalities, such as asset management, database management, selection and property management, etc. The default

interface is usually what the game looks like once released. One can also edit and manipulate objects in the default interface. Once an object is selected in the default interface, we can change its properties either from the default interface or from ParaIDE.

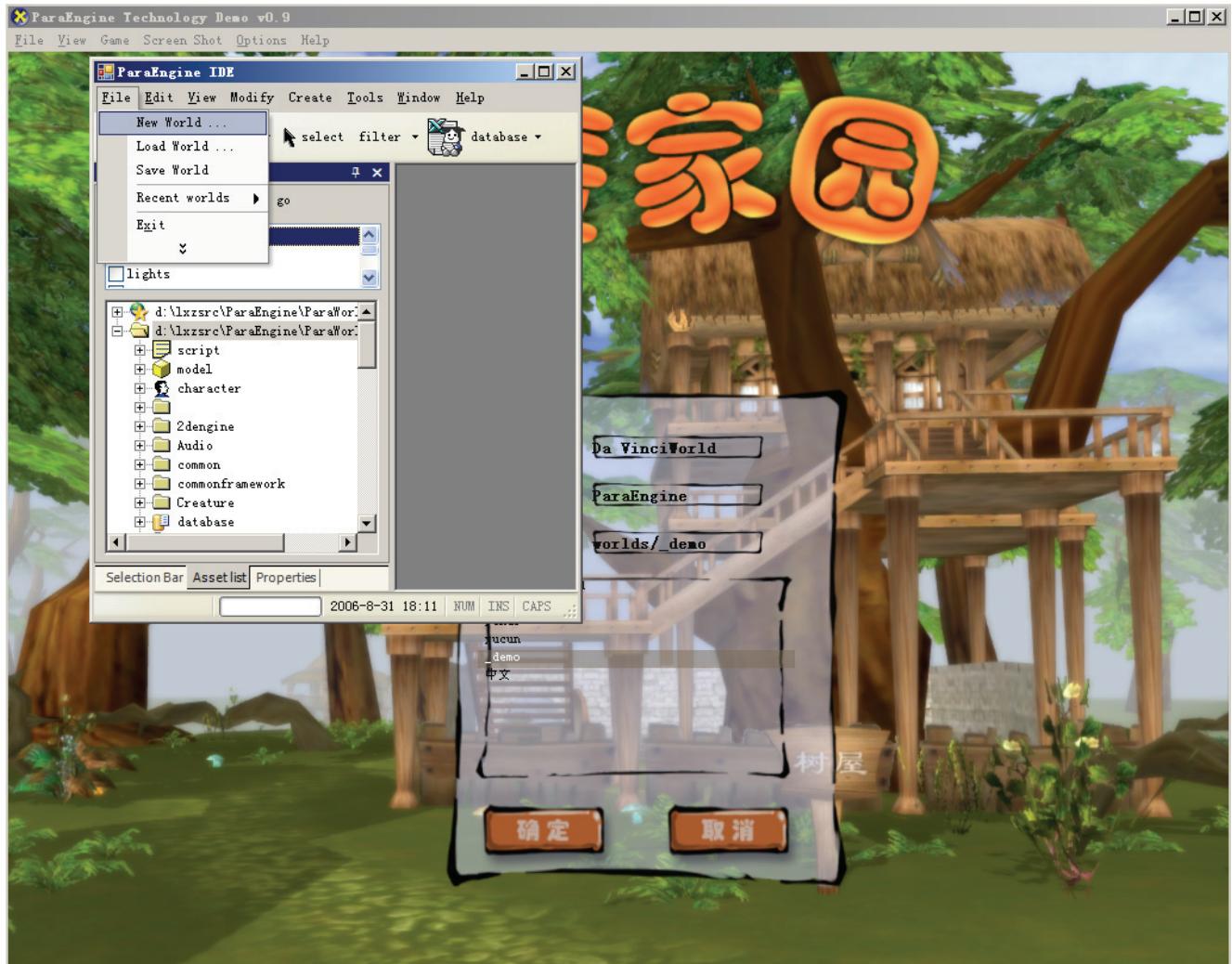


Figure 19.1 Default Interface and ParaIDE

ParaEngine SDK professional edition have a collection of API for advanced developers to write plug-ins for ParaEngine through both native C++ and .Network Framework API. In fact, ParaIDE itself is written in C# language using ParaEngine Managed API for .Net Framework. Most future plug-ins of ParaEngine for advanced developers will be released in this way. One can find standard plug-ins as DLL files in the ./plugins directory. Most plug-ins including the ParaIDE is only used by developers, and does not need to be included when shipping the game.

The scripting language or NPL is the only language that more serious developers need to learn when making games using ParaEngine. NPL is the main language for writing UI and game logics. When a game is released, all its script files must also be included. Most ParaEngine powered demos as well as the world editor (default interface) is completely written in NPL. So basically everything you see or interact in the 3D world is written in this

language. One can also load and invoke DLL plug-ins written in other languages from the scripting interface, which extends the scripting interface greatly.

Figure 19.2 shows components of ParaEngine, API and their programming language.

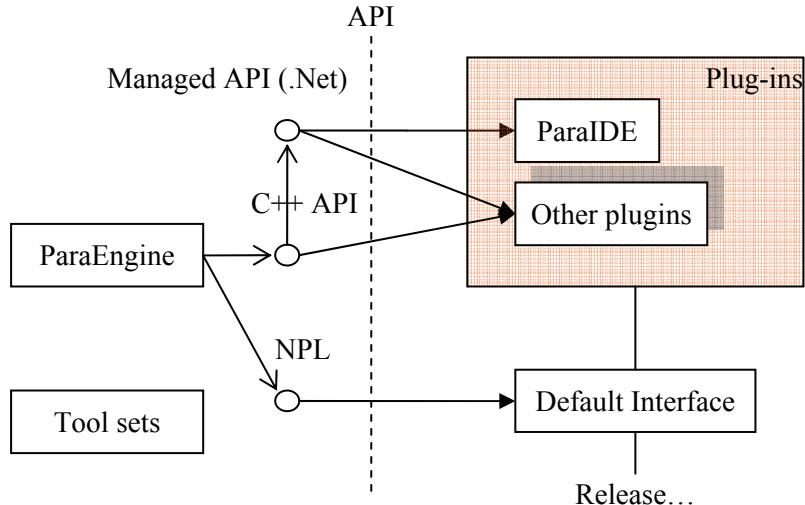


Figure 19.2 ParaEngine components, API and programming language

ParaEngine SDK also comes with a collection of tools. The most important tool is the model exporter. They allow serious game makers to import models, animations, effects, terrains, or even game scenes from professional 3D authoring environment, such as 3dsmax, to ParaEngine.

B.4 Creating a New World

A game may consist of many 3D worlds, some are huge and some are small. For this demo game, we need just one small world. Let us name it Da Vinci World. We can create it either from the default interface or ParaIDE.

In ParaEngine, an important concept when creating a new 3D world is called inheritance. When creating a new 3D world, it allows users to specify a parent world from which the new 3D world is derived (not cloned). It is similar to class inheritance in C++. When the parent world is changed, all its derived worlds also change unless the derived world has overridden or made changes to the modified portion of the parent world. In other words, the game engine automatically detects changes made to the current world, and it clones the smallest portion of the parent world if that portion has never been modified in the current world before. Figure 19.3 shows the illustration. We see that portions of the current world's game data are from its parent worlds.

The smallest pieces of overridable game data in a game world are objects in a terrain tile, terrain elevations of a terrain tile, terrain texture layers of a terrain tile and NPCs of the whole world.

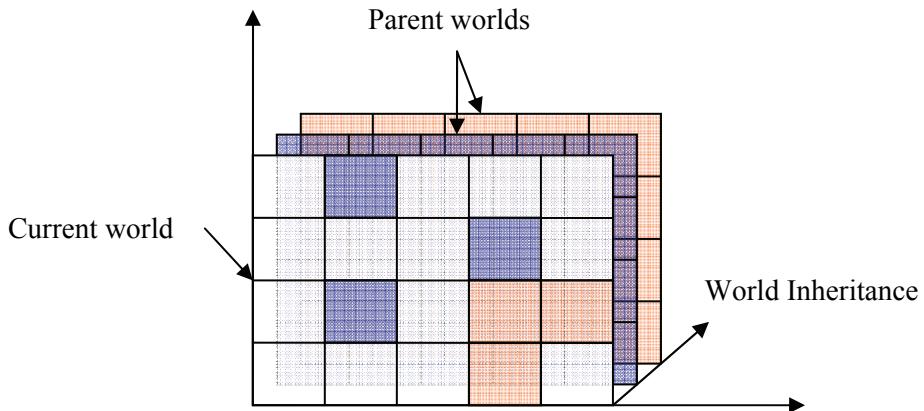


Figure 19.3 World Inheritance

Our website maintains a repository of game world templates. One can download a variety of game worlds from our website and start a new world on top of that. If one does not specify the parent world, it will derive from a default empty world with flat terrain and default of everything.

To create a new world in ParaIDE, select menu File->New World, as shown in Figure 19.4.

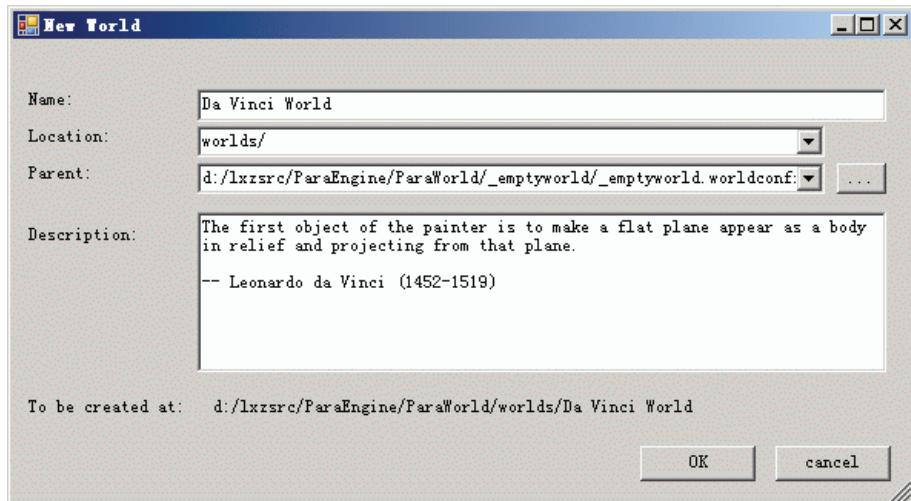


Figure 19.4 Creating a New World

All game world data are stored in a folder with the same name as the input world name. A newly derived world may only be a few KB in size. As one makes changes to it, the size of the world will grow slowly. When releasing a world, make sure that you include everything in the world folder, as well as files in the parent worlds' folder. However, if one overrides every portions of the parent world, there is no need to include the parent world any more.

B.5 Making Game Scene

A game scene comprises of terrain, models, characters, lights, and a few other global objects and effects. We can create and modify terrain, lights, and all global objects directly in the default interface. As for models and characters, one can use all models published from our website. ParaEngine Dev Studio has made these customizable models and characters in the hope that more people (young or experienced) can create high-quality game world in minutes. If you have the skills, you can follow the steps as described in section B.6, B.7 and B.8 to make or contribute your own models and characters. In this section, we assume that you are using assets released by ParaEngine SDK.

We have observed children aged 10 to create 3D scenes using the default interface, as they wander in the world. It is quite a try-and-error process, which they enjoy very much. For older people like readers of this book, we suggest you plan in advance and start from the terrain first, and then models and characters. In the game proposal section, we have described our game scene verbally. Let us review it here:

A fantasy world surrounded by mountains with water and island in the middle. The world is beautiful but looks a bit barren at first. There should be some open flat land where buildings can be built and plants can be grown.

Now, we will use a sketch of Leonardo da Vinci to give you a better illustration. Serious game studios usually draw concept arts before designers set out making the scenes.



Figure 19.5 Sketch of Leonardo da Vinci

For this demo, we need not strictly follow the concept arts. And because you are limited by the number and type of available models, you can just try your best to make it feel right. Figure 19.6 and Figure 19.7 shows sample scenes made by us using only available terrain modifiers, textures and models. It might take one to several hours to complete for first time makers. So please be as patient as children.



Figure 19.6 Sample Scene Screenshot 1



Figure 19.7 Sample Scene Screenshot 2

The following three DIY sections teach you the basics of building your own advanced terrain and models. You can escape them all and continue with the B.9 section.

B.6 Do It Yourself: Advanced Terrain

In the default interface, one can modify the terrain elevation and texture layers interactively. However, it will be slow and inefficient to make large terrains in the 3D environment of the default interface. ParaEngine comes with a terrain exporter tool for 3dsmax. It can export mesh grid from 3dsmax to “.raw” file (the height map file used by ParaEngine). The raw file is called terrain modifier file, which can be applied to the existing terrain in the game scene. Please refer to terrain chapter for more information.

The advised terrain making procedure is: first making the terrain modifiers in 3dsmax or other dedicated software and then fine-tuning the terrain in the 3D environment using the default interface.

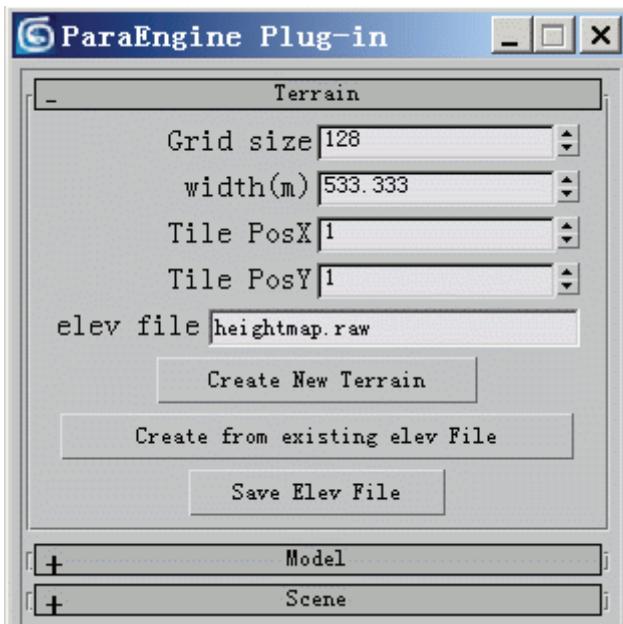


Figure 19.8 Terrain exporter in 3dsmax

Figure 19.8 shows the terrain exporter in 3dsmax. The exporter is written in 3dsmax script. One can start it in 3dsmax by running the start up script file: world editor/3dsmax plugin/Start_ParaEditor.ms

The terrain plug-in can create terrain at a given tile position in the world as well as terrain modifiers which can be applied at any location of the 3D world. We will only show how to create a terrain modifier here.

A terrain modifier is an elevation or height map file of a square grid of arbitrary size. When the terrain modifier is applied to a location of the 3D world, it will add the height map in the terrain modifier to the terrain centered at that location. The edge of the modified terrain is smoothly interpolated between the original terrain height and the height in the modifier.

To create a terrain modifier, you need to specify the grid size, such as 128; clear the text in the elev file text box to empty; and then press the Create New Terrain button. A mesh grid object will be generated. One can modify the mesh object, and then press Save Elev file.

For the default interface to automatically locate your elevation file, please save the terrain modifier file as raw file under the ./model/others/terrain/ directory. When you open the

default interface, you will notice that your newly created terrain modifier is displayed under the terrain category. To apply the terrain modifier, move the character to the desired location and then double click the modifier name. If you use the ParaIDE, you can browse to the terrain modifier file in the asset list view, right click on the file icon and select Invoke in the pop up menu. See Figure 19.9. The actual size of the terrain modifier as appeared in the 3D world is proportional to the grid size. The proportion is subject to global game world settings. A 128*128 mesh grid will match exactly to the size of your terrain tile. So if your terrain tile size is 512*512 meters, then one grid in the mesh will be 4 meters long.

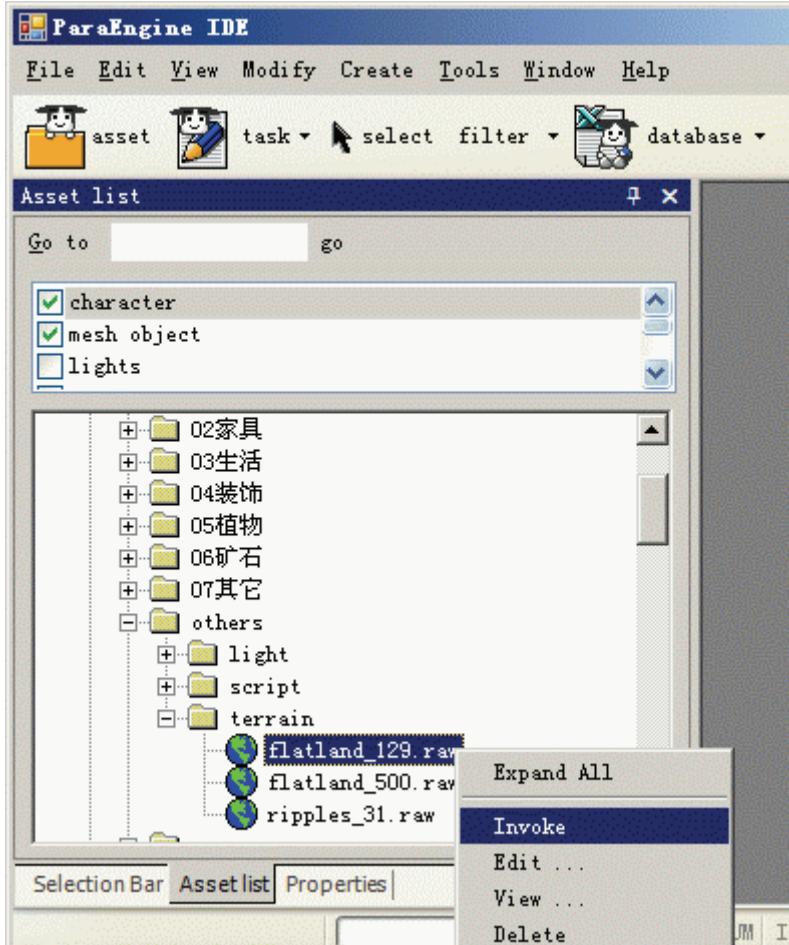


Figure 19.9 Apply Terrain Modifier in ParaIDE

ParaEngine comes with a collection of pre-made terrain modifiers. One can combine them to create versatile terrain. When the game world is saved, the elevation files of the entire world are saved as a collection of raw files. These raw files have the same format as the terrain modifier; so they can be imported to 3dsmax for fine tuning.

B.7 Do It Yourself: Static Models

To DIY your own static models is very simple if you are familiar with any of the 3d authoring environment. The static model format of ParaEngine is fully compatible with DirectX X file format. However, to access full features of ParaEngine, one needs to use the

ParaX Model Exporter plug-in provided by ParaEngine SDK. At the time of writing, ParaEngine only supports model exporters for 3dsmax version 6, 7 and 8.

Please read the ParaX Model Exporter manual for more information. When making models in 3dsmax, artists should follow the following principles:

- One generic unit in 3dsmax is one meter in ParaEngine. So it is advised that artists use meters or generic unit in 3dsmax.
- The z axis in 3dsmax is y axis in ParaEngine. Fortunately, they all mean the world up direction. Basically, what one sees in 3dsmax is the same as that in ParaEngine.
- Reset transform of the model before exporting. Please refer to 3dsmax manual for how to do it.
- All mesh faces must be assigned a standard texture material in order to be treated as visible when exported to ParaEngine.
- Objects or mesh face that does not have a material assigned to it is treated as physics face only. Physics faces will block characters and will not be rendered. We can use the 3dsMax primitives such as boxes, cylinders, etc to build the physics faces in the model.
- If there are multiple renderable meshes or objects in the scene, it is better to convert all of them to a single mesh before exporting. This can be done by first converting all objects to editable meshes, and then attaching all other meshes to a particular one. 3dsmax will automatically combine all used materials to a single multi-material, which can be exported by the exporter.
- For ordinary textures with or without transparency, one can use standard material and specify only the diffuse and opacity maps. If the texture is opaque, only specify the diffuse map and leave the opacity map empty. For texture with transparency, specify both the diffuse and opacity map. Please note that the diffuse and opacity map should be the same texture file, where the opacity is read from the alpha channel of the bitmap. Material that only uses diffuse map will be exported as DXT1 format which contains 1 bit alpha; material with opacity map will be saved as DXT3 or DXT5 which has 3 bits alpha. Artist is advised to use "tga" file format during exporting. The ParaX Exporter will automatically save all used texture files as "dds" format according to their usage.
- Texture UV: Never set texture tiling in the material editor, since they are not exported by the exporter. Instead, use UV mapping or UVW unwrapping in the 3dsmax modifier stack.
- File path in ParaEngine are all relative to the root directory. Absolute file path is still supported, however, when deploying the game to a different location, the file will not be found. Hence, it is highly advised that one put the max file and its referenced textures in the same folder, or at least under the same root directory. The root directory of a given file is defined as closest parent directory that contains the *ParaEngine.sig* file. If one looks at the ParaEngine SDK installation folder, it will find the sig file there.
- Reflection map: specify ray-tracing in the reflection map of 3dsmax to export reflective surface. Currently, only horizontal reflective surfaces at y=0 plane is supported.

- Cubic environment map in 3dsmax is supported: Environment mapping: specify reflect/refract map in the reflection map of 3dsmax material to export environment mapped surfaces. The exporter will use the 6 surface files to generate the exported cubic texture file. The exported cubic texture file has the same name as the diffuse map plus "Env". For example, the diffuse texture is "face.dds", then the exported environment map will be "faceEnv.dds"
- Light map must use a separate UV set and is specified as a bitmap in the reflection map channel of 3dsmax. The exported texture file has the same file name as the specified bitmap file. For example, if the reflection texture is "face.tga", the light map will be "face.dds".
- The light map intensity can be specified by the amount property of the reflection map. The amount parameter in 3dsmax is [0,100]. The actual light map intensity exported is X/10, with the exception that 100 stands for 1. Hence, 10 is 1, 20 is 2, 99 is 9.9; yet 100 is 1. light map may be shared among multiple materials
- If the texture file name ends with _a{0-9}{0-9}{0-9}.xxx, it will be regarded as an animated texture sequence. Always specify the last texture in the sequence, such as "smoke_a120.tga". All texture files can be animated, such as diffuse and light maps.

It is advised that you browse the sample 3dsmax source files under the SDK's model directory before exporting your first model. Once you are satisfied with your model in 3dsmax, you can export it by selecting File→Export (ParaEngine ParaX file). There are not many options and you do not even need to specify a valid file name. The ParaX Exporter will guess most of the options for you as well as the file name and location to be saved. After examining the guess report, click OK and it is done. See Figure 19.10

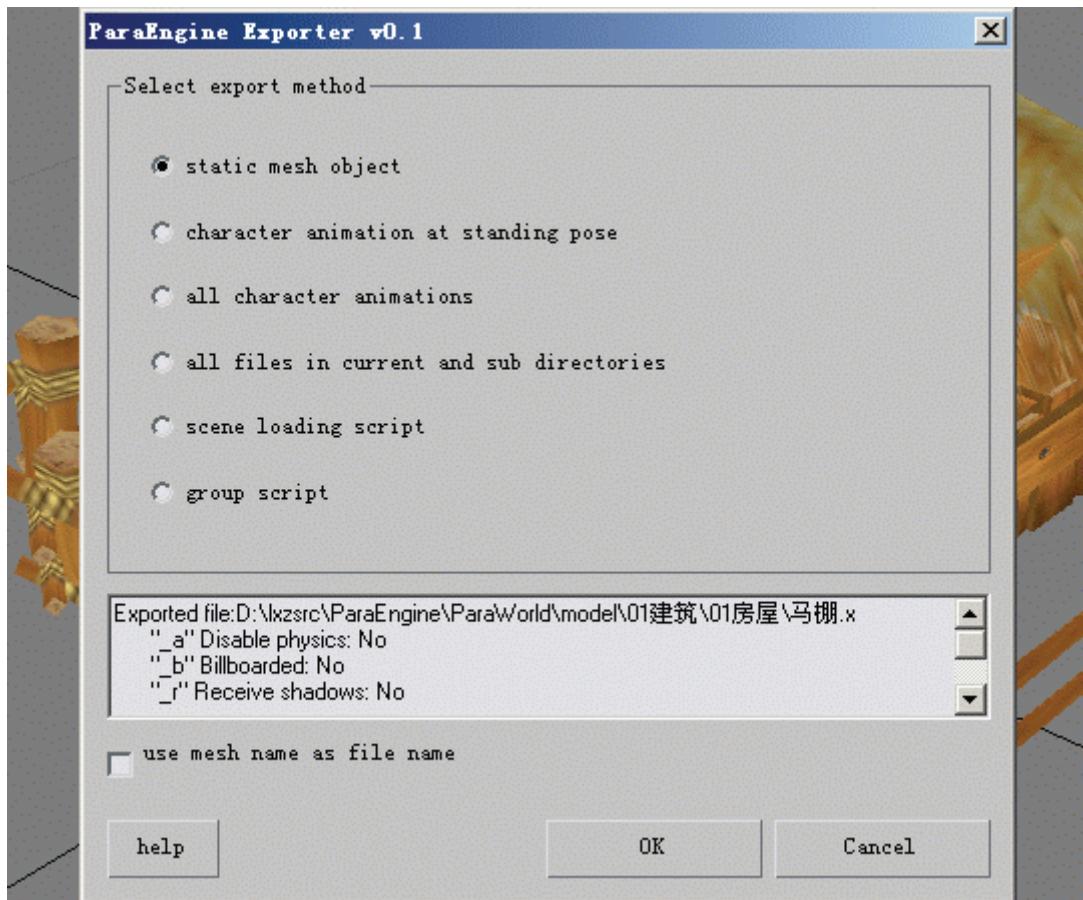


Figure 19.10 ParaX Model Exporter

To view your models in the game engine, you can create an instance of the model using either the default interface or the ParaIDE. To create a model in the ParaIDE, browse to your model file in the asset list, and double click the file icon to create a model instance at the current player's position.

B.8 Do It Yourself: Characters

Characters and animated models are a little bit more difficult to make. One needs to follow more strict rules in order to correctly export animated models.

First of all, characters may have one to several hundreds of animations. There is a default animation called standing animation which all characters must provide. Secondly, all animations of the same character must share the same bone structure as well as bounding weights or physiques. Thirdly, some animations are looped and some may have a global speed. Each character animation should be stored in a separate file. All character animation files must be put in the same directory. A character model directory contains only models files which share the same mesh and bones.

When making characters with multiple animations, we usually start from the character with the standing animation. In order for ParaX model exporter to guess the correct options, we need to name each 3dsmax source file according to the following specifications.

```
sObjectName (nAnimID [, fAnimSpeed], [, nonloop]).max
```

sObject Name is arbitrary. For example, "MyModel(0)" or "MyModel(4, 1.234)" or "MyModel(30, 0, nonloop)".

nAnimID: this is the animation id for the max file. ParaEngine reserves animation id in the range [0,255]. Animation ID outside this range is user defined animation sequences. The mapping from id to their interpreted action can be found in the animation ID table of the exporter manual. For example, 0 means default standing animation; 4 means walking animation.

fAnimSpeed: This specifies animation speed in meters per second unit. Default value is 0.

"nonloop": if "nonloop" is present in the file name, the animation exported will be non-looped, otherwise it is saved as looped animation.

For example: suppose you have the following max and texture files in the ./dog/ directory

```
./dog/dog (0).max  
./dog/dog_walk (4, 2.35).max  
./dog/dog.tga
```

You can open ./dog/dog(0).max in 3dsmax, and select File→Export (ParaEngine ParaX file), and then type anything as the file name. When the export dialog pops up, you will see that the exporter has guessed most of your exporting options. You just need to select whether you want to export a single animation for testing or all animations in the current directories. If your models are OK, the exporter will generate a file at "./dog/dog.x", which contains your current animation or all animation sequences. This may takes some minutes if the character has many animation sequence files.

It is advised that you browse the sample 3dsmax source files under the SDK's character directory before exporting your first animated model.

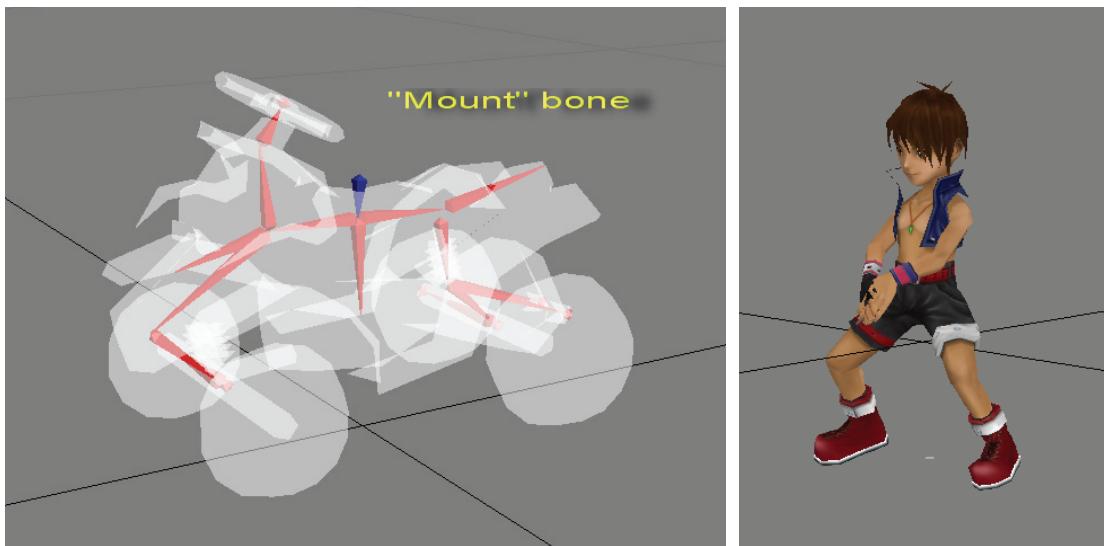
B.8.1 Special Rules

There are more restrictive rules when making animated model than the static ones. If you do not follow the rules, the exported model will not look right in the game engine. Some of the common rules are given below.

- All meshes and bone structures of the character should be the same in all character animation files. It is advised that one copy and paste the standing animation file when making other animations.
- The character's facing should be the positive x axis in 3dsmax.
- The following animation method is supported: skin, physique, biped (motion capture), simple key transform. Please note that only one of these methods can be used for a single character.
- Before binding bones to a mesh, reset the mesh's transformation.

- Avoid using scale animation. If one has to, use uniform scale only, i.e. scale the same amount along x, y, z axis. The current exporter ignores the scale's rotational axis.
- Floating bones or floating meshes are not supported. In other words, all bones must have a single parent bone, and if a mesh is animated all of its vertices must be bound to at least one bone.
- When using biped to animate characters, make sure that the initial pose of the character is the same as the pose of the character when the skin modifier is applied.
- We allow models to be attached to other models, such as character body attachments or mount system. These are realized by adding special bones to the character. These special bones are converted to attachment points when exported. A bone whose name contains the following string will be regarded as attachments.
 - "Head" or "头部": Up to 4 parent bones of the head bone is used to rotate the character's upper body.
 - "L UpperArm": left shoulder where you can attach armory.
 - "R UpperArm" right shoulder where you can attach armory.
 - "L Hand" left hand which can hold other object models.
 - "R Hand" right hand which can hold other object models.
 - "Mount" or "Shield" the mount position for car or horse, etc. It may also be the shield position where weapons may be attached on character models. "Mount1" and "Mount2" are reserved in case that there multiple mount attachment points.

For example, if a car character defines a Mount bone at the seat position, and another human character has a mount animation sequence, then the game engine can automatically mount the human character to the car at runtime. Figure 19.11 shows animated model files in 3dsmax and Figure 19.12 shows the animation result in the game engine.



Character with “Mount” bone

Character with mount animation

Figure 19.11 Mount Bone and Mount Animation



Figure 19.12 Mount one character over another

B.9 Adding Artificial Intelligence

Now we will add some AI to the game world, making it alive. Only character objects can have AI. There are some pre-made AI templates which can be assigned to the currently selected character, with a single click of the mouse in the default interface.

B.9.1 Do it yourself: Artificial Intelligence

To build your own AI, you need some knowledge about how characters are simulated in the game engine. Please refer to the AI chapter of the book for more information. AI in ParaEngine is layered; only the top layer can be edited completely via the scripting interface. The following call back events can be registered per character.

On_Load	Called when the character is loaded from disk or database to the scene. Although it is not called by the character simulator, it looks and functions like a simulation event.
On_EnterSentientArea	When other game objects of a different type entered the sentient area of this object. This function will be automatically called by the character simulator.
On_LeaveSentientArea	When no other game objects of different type is in the sentient area of this object. This function will be automatically called by the character simulator.
On_Click	When the player clicked on this object. This function will be automatically called by the character simulator.
On_Perception	When other game objects of a different type entered the perceptive area of this object. This function will be automatically called by the character simulator.
On_FrameMove	This function is called every frame when the character is sentient. This is mostly used by AI controllers.

The event handler registrations can be done in both ParaIDE and the default interface by editing the character's property. When a new character is created, all of its event handlers are empty. When you try to add a new event handler, you can either specify an existing handler from a script file, or let the game engine create a new empty handler for you. Figure 19.13 shows the event handler properties of a character in the default interface.



Figure 19.13 Event handler properties in the property window of default interface

After clicking the open button next to the property field, a new script with the same name as the character will be automatically created in the ./character/ directory of the current world, and opened in the default script file editor as shown in Figure 19.14.

```

1 --[[ a character event handlers
2 author:
3 date:Fri Jun 09 16:49:32 2006
4 ]]
5 if(not _character) then _character={} end
6 if(not _character.a) then _character.a={} end
7
8 function _character.a.On_Load()
9   local player = ParaScene.GetObject(sensor_name);
10  if(player:isValid() == true) then
11    -- TODO: do your work here
12  end
13 end
14
15 function _character.a.On_EnterSentientArea()
16 end
17
18 function _character.a.On_LeaveSentientArea()
19 end
20
21 function _character.a.On_Click()
22 end
23
24 function _character.a.On_Perception()
25 end
26
27 function _character.a.On_FrameMove()
28 end
29
30 function _character.a.On_Event()
31 end

```

Figure 19.14 Empty event handlers script file in the editor

When any event handler is called, a parameter called `sensor_name` contains the character's name. We can get a character instance of the character by calling the following script command.

```
local player = ParaScene.GetObject(sensor_name);
```

We can access to all other perceived objects of the current character as shown in the example script below.

```

function _character.a.On_Perception()
-- do your AI code here.
local player = ParaScene.GetObject(sensor_name);
if(player:isValid() == true) then
  local nCount = player:GetNumOfPerceivedObject();
  for i=0,nCount-1 do
    local gameobj = player:GetPerceivedObject(i);
    if(gameobj:DistanceTo(player) < 5) then
      log("Saw an object");
    end
  end
end
end

```

It is a good practice to use “`sensor_name`” parameter to get the current character name instead of hard-code the character name in script. This allows the same event handler to be reused by multiple characters of the same type. However, in case a character handler is only used by a unique character in the game, one can directly use the character name in script.

The event handler registration information as well as the memory states of a persistent character are stored in the NPC database under the current world's directory by default. We can view this database using ParaIDE. Just browse to the world directory in the asset list, select the database file and double click its file icon. See Figure 19.15.

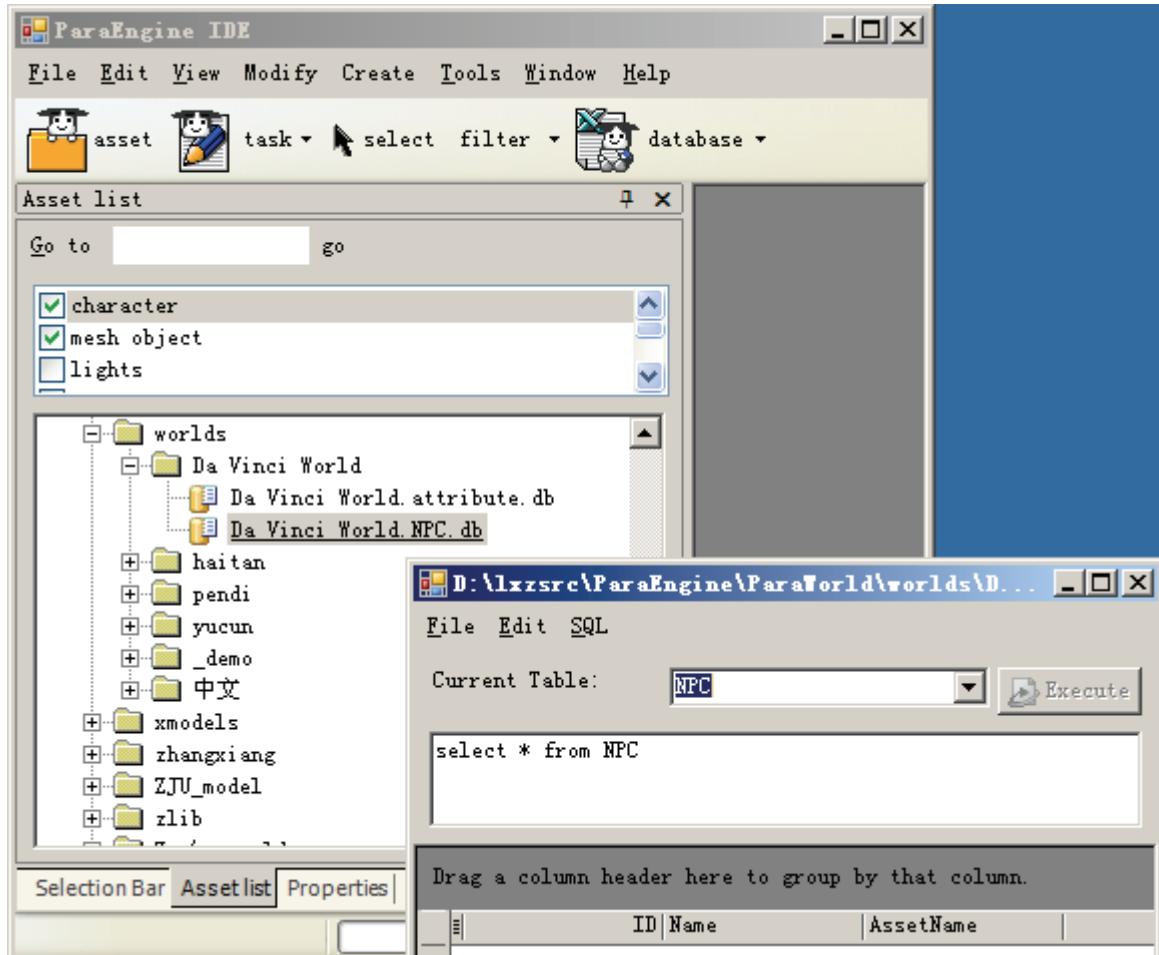


Figure 19.15 NPC database in ParaIDE

It is advised that you see the sample AI templates written by us under the script/AI/ directory of the ParaEngine SDK. Also NPL programming reference is a must read for any one who wants to use the NPL language.

B.9.2 Do it yourself: Using AI Controllers

Instead of controlling the character each frame in the event handler, we can assign AI controllers to the character. We can regard AI controller as a certain logic unit. When they are initialized and assigned to a given character, it will automatically control the behavior of the character from that time on. This saves us development time and makes the AI logics clearer. Because AI controllers are mostly written in native C++ language, it runs faster than script instructions. Another advantage of using AI controller is that some pre-made tools are available to quickly customize a certain AI controller. For example, one can use the sequence AI controller to instruct the character to follow a certain path, where the path can be specified using a pre-made tool with graphic user interface.

The following are some code example of using three different kinds of AI controllers. Please note that various AI controllers can be combined to generate complicated AI behaviors.

Face Tracking Controller

Face tracking controller controls a character to always face a given target or another character. Face tracking controller has relatively high priority. This controller is commonly used when a player approaches an NPC. The NPC with a face tracking controller will automatically rotates its neck to face the incoming player.

We can assign and release this controller at any place and at any time. For example, we can assign it in the On_Load() event handler, so that the character will always have this facing behavior when it is loaded. See below. The bold text enabled the face tracking controller.

```
function _character.a.On_Load()
    local player = ParaScene.GetObject(sensor_name);
    if(player:isValid() == true) then
        local playerChar = player:ToCharacter();
        playerChar:AssignAIController("face", "true");
    end
end
```

Follow Controller

Follow controller controls a character to follow another character as long as the target is in sight. The following script instructs the character to follow another player named “girlPC”.

```
function _character.a.On_Load()
    local player = ParaScene.GetObject(sensor_name);
    if(player:isValid() == true) then
        local playerChar = player:ToCharacter();
        playerChar: AssignAIController("face", "true");
        playerChar: AssignAIController("follow", "girlPC 2.5 -1.57");
    end
end
```

The two additional parameters following the target player name is radius and angle, which specify the desired position of the following player around the target player. Figure 19.16 shows the meanings of the two parameters. So the character in the example code will try to maintain a distance of 2.5 meters from the target player and stay to the left of the target.

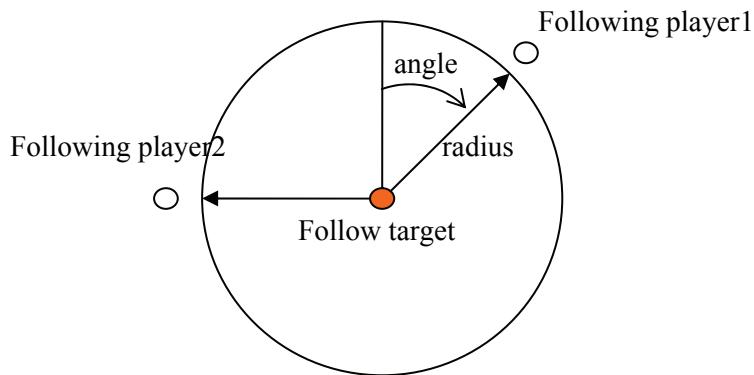


Figure 19.16 Follow controller parameters

Sequence Controller

Sequence controller controls a character according to some predefined sequenced commands list. It executes at most one command per frame, and it will only execute the next command when the current command is finished. One can think of a sequence controller as a short computer program. The program may contain branch and loop operations.

Sequence controller can be combined with character sensor events to model fairly complex AI logics. A sequence controller alone can be used to model behaviors such as patrolling guards, wandering enemy creatures, busy citizens on a city street, etc.

```
function _character.a.On_Load()
    local player = ParaScene.GetObject(sensor_name);
    if(player:isValid() == true) then

        local playerChar = player:ToCharacter();
        playerChar: AssignAIController("face", "true");

        -- a sequence can be read from file or database; here I just hard-coded them.
        local s = playerChar:GetSeqController();
        s:BeginAddKeys();
        s:Lable("start");
        s:PlayAnim("EmotePoint");
        s:Wait(3);
        s:Turn(0);
        s:WalkTo(10,0,0);
        s:Wait(3);
        s:Turn(-1.57);
        s:RunTo(0,0,10);
        s:Turn(3.14);
        --s:Pause();
        s:WalkTo(-5,0,0);s:Jump();s:RunTo(-5,0,0);
        s:Turn(1.57);
        s:MoveTo(0,0,-10);
        s:Goto("start");
        s:Exec(";NPC_SEQ_EXEC_Test()");
        s:EndAddKeys();
    end
end
```

```

function NPC_SEQ_EXEC_Test()
    local player = ParaScene.GetObject(sensor_name);
    if(player:isValid() == true) then
        log("this is from sequence controller of character"..sensor_name);
    end
end

```

The above sequence program tells the player to walk around a rectangular path repeatedly, while performing some animations. Please refer to the NPL programming reference for exact meanings of sequence commands.

All way points in sequence controller are specified in relative position, so that the same sequence controller may be reused for different characters.

B.10 Do It Yourself: User Interface

Your user interface stuff is written in NPL scripting language. This section does not teach you how to use this language. It only suggests some basic ideas of how the existing SDK script files are organized and how to change the user interface by modifying these files.

B.10.1 Game Loop File

First of all, the first script file which will be executed when the application starts is ./script/gameinterface.lua. This script file is also the default game loop file, which will be executed several times per second.

Basically, the first thing that you want to do in it is to change the game loop file to a new one of your choice. We can do it by calling.

```
ParaGlobal.SetGameLoop("(gl)script/kids/main.lua");
```

After calling this function, the “gameinterface.lua” will never be called again. Instead the newly specified file (in the example, “script/kids/main.lua”) becomes the current game loop file, which will be executed several times per second. It is in this script that you create those first user interfaces in your game.

B.10.2 UI Controls

UI controls are basically created in a similar way like in other windows programming language. For example, a container and a child button control can be created as below.

```

local window, button, text; --declare local variables
--create a new window called "mainwindow" using left top alignment at (50,20) with size 600*400
window=ParaUI.CreateUIObject("container","mainwindow","_lt",50,20,600,400);
--attach the UI object to screen (root)
window:AttachToRoot();
--create a new button called "btnok" at (50,350) with size 70*30
button=ParaUI.CreateUIObject("button","btnok","_lt",50,350,70,30);
--attach the button to the window
window:AddChild(button);
--set text to be displayed on the button
button.text="OK";
--if the button is clicked, specify the event handler
button.onclick=";ParaUI.Destroy(\"mainwindow\");";

```

B.10.3 IDE libraries

ParaEngine SDK comes with many useful script files for composing and editing the 3D environment and 2D user interface. These script files are under the `./script/IDE/` directory. For an example of using these IDE libraries, one can refer to script files under `./script/demo/` and `./script/kids/`. The “demo” folder contains the source code for the default interface’s main menu bar and all of its function modules, such as object creation and editing, terrain and ocean editing, etc. The “kids” folder contains the source code for everything else in the default interface.

B.11 Release

There are a number of ways to release the sample game. If you only used the default interface and ParaIDE to build the game, you can just release all files under your world folder. People who have an instance of ParaEngine can open your world and play in it. However, if you used any of the stuff in the do-it-yourself sections, you will also need to include all model and script files besides the files in the world folder. The most complete way to release a game is to include everything under the ParaEngine install directory.

B.12 Conclusion

Making a game can be as easy as playing it. However making serious game from the ground up can be a daunting task. The design principle of ParaEngine is to simplify game making so much that anybody with or without former computer experience can use it to create interactive 3D worlds. For intermediary and advanced users, it gives them access to the deep parts of the game engine, and facilitates them to create resources that could be shared by other developers.

Appendix C

About ParaEngine

C.1 Introduction

ParaEngine is a distributed 3D computer game engine. It aims to develop the next generation massively multiple-player online game, where the game world content and logics are distributed over arbitrary networks. Neural Parallel Language (or NPL) is an extensible programming language which separates code logics from its hardware network environment. It is a new programming standard designed to support (1) frequent code updates, (2) complex graphic user interface and human-computer interactions, and (3) reconfigurable code deployment in a distributed network environment. The language is tightly integrated with ParaEngine.

C.2 ParaEngine Specifications

The following table shows the features of ParaEngine.

Table 1. ParaEngine Specifications

Supported OS:	Windows XP/2000/2003 server
API:	DirectX 9.0C
Language:	C/C++ (VS.NET 7.1)
Features: Graphic engine:	<ul style="list-style-type: none">- 3D scene management and rendering system, supporting 40,000m*40,000m continuous game world. It is suitable for RPG games with super large continuous map, many globally movable characters, complex scene triggers and story design.- Advanced programmable pipeline for all renderable objects in the scene.- Flexible shader management on a per object basis. One can dynamically change the shader program of any scene objects on a per frame basis.- Fixed function pipeline support, which allows the game engine to run on older machines while retaining most of the graphics qualities. It can be mixed with the programmable pipeline to adapt to a wide variety of graphics cards on the market.- Hardware-occlusion testing can be turned on in the rendering pipeline, if there are a huge number of small yet dense models in the game scene.- Tile based ROAM/CLOD terrain engine. Because it is tiled, there is no limit to the size of the terrain. Terrain holes can be created dynamically, which allows game developers to build caves, tunnels or underground world directly in the same global terrain environment. In-game terrain editing supports height field modifications, unlimited multi-texturing on terrain surface, auto-breaking super large terrain texture files in to smaller ones, fast ray-tracing on global height field.- FFT based ocean wave simulation with reflection and refraction mapping for both above and underwater scenes. Supporting real time shorelines and underwater scene blur effect.- Unlimited ranged light sources can be put anywhere in the game world.- Robust shadows: supporting both shadow volume and shadow mapping at the same time.- Game resource management system: textures (dds, png, tga, bmp, jpg), 3D objects (x file, m2, ParaX file), 3D skeletal animation (X file, m2 and ParaX file), effect file (FX file), sound (wav, mp3), virtual file management (zip), database file management.

- A comprehensive 2.5D GUI engine, fully scriptable through the NPL language. Drag and drop controls, scrollable containers, three-state buttons, list box, slider bar, grid view, AVI video player, edit box, IME edit box supporting various language input methods, supporting GUI objects attached to 3D scene objects.
- Skeletal animation with customizable character appearance and reconfigurable equipment, such as hair style, skin color, clothes, etc. Settings can be made persistent by the pre-designed database. Supporting BVH motion capture export for biped based skeletons.
- Particle systems are supported in the animation systems
- Three customizable follow cameras modes and one free camera mode. The camera will automatically position itself to avoid collision with the view frustum and the physical environment. Thus, it is eligible for third-person RPG games.
- In-game movie recording in any movie format, such as AVI.
- Fast mouse ray-picking with 3D scene objects.
- [Effect] global sunlight simulation. It will affect shadows and scene illumination.
- [Effect] per-pixel lighting and fog can be turned on in each model shader.
- [Effect] For each renderable object, some other effects can be dynamically applied to it when necessary, such as object construction shader, occlusion shader, shadow mapping shader, etc.
- [Effect] Cubic environment map, animated texture, reflective texture, light map on a separate UV set, all of which can be exported to the game engine.

Scripting engine:

- The build-in scripting engine is powered by NPL language. It mimics the functioning of neural networks and codes the logics of distributed game world into files that could be deployed to arbitrary runtime locations on the network.
- In the NPL programming environment, it makes little difference between developing network applications and stand-alone ones.
- All API of the game engine are exposed through the scripting interface, including GUI, game events, 3D scene management, AI game logics, resource management, networking, database queries, etc. It is possible to develop a traditional RPG game through the scripting interface only. The scripting interface is well documented with over 20,000 lines of source code examples.
- The new scripting paradigm implies new ways of game content development on the following aspects: online game and game society establishment and maintenance, non-linear stand-alone AI behaviors and networked AI behaviors, stand-alone game story development and network distributed story development, game cut-scene design and interactive game movie shooting, game map design/storage/transmission, visual scripting environment, etc.
- All network behaviors are written in NPL.

Middleware support:

- Polygon level physics effect middleware support [optional]: Novodex [Havok, ODE]
- Vegetation middleware support [optional]: [speed tree]

AI, physics, and others:

- Fast character simulation engine. It supports a large number of characters on a vast continuous game world (as long as 32bits floating point position vector do not lose accuracy). It integrates well with the global terrain and ocean engine. It provides interfaces for path-finding, perception, collision detection and response. It supports ray picking based character controller. For each character, quite a few character sensors can be turned on to process custom character AI logics in the NPL script callback functions.
- AI controllers: multiple hard-coded AI controllers can be parameterized and combined with each other or with character sensor scripts to efficiently emulate convincing AI behaviors. Supported AI controllers are: (1) Movie controller: it can record and replay character actions in a time accurate manner. This is an easy way to build in-game

character cinematic. (2) Sequence controller: a list of AI commands to be executed one or several per frame. It can be used to perform general NPC logics such as patrolling, dialoging, etc. (3) Combat controller: such as attack, evade, flee actions. (4) Face tracking controller: rotate the character's neck to face a target. (5) Follow controller: follow another object.

- Biped state manager. It is a finite state machine which applies jumping, walking and swimming animation, etc and their transitional animations automatically, according to the physical environment.
- Mount system: characters can be mounted on other characters, such as a person being mounted on a horse.
- Local database support, supporting most SQL-92 standard.
- Debugging and logging can be turned on which tracks all aspects of the game engine status.
- [Game object] missile object. Firing a missile to any place in the game world.
- [Game object] D&D based RPG character object. Character attributes are exposed through the scripting interface and character data is made persistent by the database. This is a helper game object for building simple RPG game.

Tools and libraries:

- 3dsmax7&8 model exporters: both static and animated models are supported. Characters with multiple animation sequences are also supported.
- 3dsmax terrain, scene exporters: this is written in 3dsmax script, and we allow users to modify the source code to suit their needs in their own specific level design tools.
- Virtual world builder: This is a simple to use in-game game development environment. It comes with a free collection of game art data. There are no sophisticated interfaces in it; even children can learn to play with it. One can create a huge 40,000m*40,000m game world by just mouse clicking and editing simple game events (the latter is only for serious users). Editable entities in the world builder includes: terrain, ocean, models, characters, GUI, sounds, lights, scripts, etc. It also supports a powerful feature called *world inheritance*, in which a new world can be built by inheriting from multiple existing game worlds. Virtual world builder is written entirely by our proprietary NPL scripting language and we allow users to modify the source code to suit their needs in their own specific level design tools.
Chinese version only. English version is planned at the end of 2006.
- World asset manager: managing resources used in a game title. Source code in C#
- In-game movie recorder: with source code in NPL
- In-game GUI IDE: with source code in NPL
- Third party tools: virtual file browser, deep exploration (3D model viewers), sqlite analyzer (database query builder), ultra edit (optional script editor).
- The game engine functionalities can be extended or customized through three set of API: (1) NPL scripting system (2) C++ API (3) .Net Framework API.
- ParaIDE: another IDE of the game engine completely written in C# using the ParaEngine .Net Framework API. Source code available. It features resource management, object property editing, game world management, client database management, etc.

Documentation and support:

- NPL scripting reference: (150 pages) *English version only.*
- Demo “Parallel World”: It is a distributed game, built on top of ParaEngine. Player may have its own game world hosted like a personal website on its PC or other web servers. *It is available on our website at the end of 2006.*
- Artists’ guide: (50 pages) both Chinese and English version.
- Book “Design and Implementation of a Distributed Game Engine”: (approx. 350 pages) written by the main author of ParaEngine. Available in print in September 2006. *English version only. Chinese version is planned at the end of 2006.*
- ParaEngine design documentation: (approx. 2000 pages) only available for enterprise edition users. *English version only.*

- Website forums: <http://www.paraengine.com>

Summary: ParaEngine is a distributed 3D computer game engine. It aims to develop the next generation online games, where the game world content and logics are distributed over arbitrary networks.

License: To be released in three editions:

- **Personal Edition:** (*Free of charge*) Users can take the full advantage of NPL scripting language to construct their own virtual game world. The following is also available: source code of virtual world builder written in NPL language, documentation and tutorials of NPL, all ParaEngine related tools and libraries. It can be used for non-commercial uses and *commercial uses with some limitations*. Available in September 2006.
- **Professional Edition:** The personal edition plus partial source code of the game engine (including all tools' source code) and technical support. **Available now.**
- **Enterprise Edition:** The professional edition plus source code of the game engine and technical support. We can also modify the game engine according to your needs. **Available now.**

C.3 About ParaEngine Dev Studio

ParaEngine Tech Studio (P.E.) focuses on the research and development of a distributed computer game engine called ParaEngine. Based on our proprietary game engine technologies, we are building various applications ourselves and we are also working with licensed clients who use ParaEngine in their own projects. We believe that game technology is the driving force to a new 3D Internet or web 3D. We would like you to join us or cooperate with us to build innovative games or other 3D applications. Please contact me directly by email: lixizhi@yeah.net. Also please visit our website at www.paraengine.com for more information.

Appendix D **Figures in the Book**

Figure 1.1 Overview of the major engine modules	7
Figure 2.1 Basic Skeleton of a Computer Game Engine	11
Figure 2.2 Spiral development path.....	12
Figure 2.3 Timing and I/O in ParaEngine	16
Figure 2.4 Asset entities in ParaEngine.....	23
Figure 3.1 File relationships in ParaEngine.....	31
Figure 3.2 File relationships in ParaEngine.....	32
Figure 3.3 Grid-based virtual world partition.....	33
Figure 3.4 Alpha blending of multiple textures on the terrain surface.....	35
Figure 3.5 From On Load Script to 3D scenes	36
Figure 4.1 Quad-tree of Terrain Tiles.....	45
Figure 4.2 Objects on Quad-tree.....	46
Figure 4.3 Object-level Testing Using Oriented Bounding Boxes.....	48
Figure 4.4 The Collaboration diagram for the root scene object.....	50
Figure 4.5 The base class and bounding volume functions	51
Figure 5.1 Typical rendering pipeline in 3D engine.....	57
Figure 5.2 Object Level Culling: object visible distance function	58
Figure 5.3 Hardware Occlusion Testing using bounding box	60
Figure 5.4 Effect Files in a Game Engine	63
Figure 5.5 Effect handle and it association with scene objects	64
Figure 5.6 Hardware Occlusion Testing using bounding box	71
Figure 6.1 Particle rendering with approximated facing using center of mass.....	80
Figure 6.2 Particle Systems in Games	80
Figure 6.3 Weather system	81
Figure 7.1 Ray/AABB Collision Testing: projection to x plane	84
Figure 8.1 Multiple ray-casting collision detection	97
Figure 8.2 Ray casting based sliding wall	99
Figure 8.3 Collision detection and response results	100
Figure 9.1 Terrain Grid.....	116

Figure 9.2 Alpha blending of texture layer.....	117
Figure 9.3 Brutal Force Terrain Triangles.....	118
Figure 9.4 Chunked LOD terrain triangulation	119
Figure 9.5 Geomipmapping terrain triangulation	120
Figure 9.6 ROAM terrain triangulation.....	121
Figure 9.7 Holes on terrain surface	122
Figure 9.8 Collaboration diagram for Terrain Object.....	122
Figure 9.9 Bounding Box and Triangles in Terrain quad tree.....	124
Figure 9.10 Repair cracks in terrain triangles.....	126
Figure 9.11 Terrain texture samples	127
Figure 9.12 Crack repair for latticed terrain	129
Figure 9.13 Degenerate Triangles	133
Figure 10.1 Reflection and refraction of light ray	141
Figure 10.2 Camera reflection.....	143
Figure 10.3 Original camera image (left) and reflection map (right).....	143
Figure 10.4 Normal correction	144
Figure 10.5 Above and under water scene without refraction map	150
Figure 10.6 Shore lines.....	151
Figure 11.1 Shadow mapping.....	154
Figure 11.2 A 3D scene rendered with shadow mapping in ParaEngine	155
Figure 11.3 1024*1024 sized shadow map generated from the light's perspective	155
Figure 11.4 Shadow volume rendering results	156
Figure 11.5 Shadow Volume Calculation for Directional Light	158
Figure 11.6 Z-Fail algorithm testing.....	159
Figure 11.7 Light map basics	163
Figure 11.8 Light map example in ParaEngine	164
Figure 11.9 Billboaded trees.....	170
Figure 11.10 Reflective surface.....	171
Figure 11.11 Local Light and render pipeline	172
Figure 11.12 Local lights example in ParaEngine.....	172
Figure 11.13 Full screen glow effect.....	174
Figure 11.14 Intermediary images in full screen glow effect.....	175
Figure 11.15 Under water effect using full screen glow	179

Figure 11.16 Fog and Sky using Programmable Pipeline	181
Figure 11.17 Fog color blending	183
Figure 11.18 Fog implementation result.....	183
Figure 11.19 Fog with large mesh object	184
Figure 12.1 Bone, skin and bone hierarchy	188
Figure 12.2 Blending poses and weight choice	191
Figure 12.3 Motion Blending Sample	193
Figure 12.4 Relationships between character animation modules.....	197
Figure 13.1 Bone Matrix and its graphical presentation in 3dsmax	208
Figure 13.2 Initial bone pose and initial mesh pose	208
Figure 13.3 Bone pose at frame N.....	209
Figure 13.4 Animation Work Flow	215
Figure 13.5 Model Export Options.....	217
Figure 13.6 Model Data Presentation	218
Figure 13.7 Script based ParaEngine plug-in for 3dsmax	224
Figure 14.1 Sentient radius and swept region.....	230
Figure 14.2 Frame rate and priority of AI modules.....	235
Figure 14.3 IGameObject inheritance graph	237
Figure 14.4 AI controller inheritance graph	237
Figure 14.5 Collaboration diagram for CBipedController	239
Figure 14.6 Collaboration diagram for CAIModuleNPC	239
Figure 15.1 Camera Occlusion Constraint and Physics.....	251
Figure 15.2 path-finding: adding dynamic waypoints.....	255
Figure 17.1 Game file system overview	276
Figure 17.2 Zip archive format overview	278
Figure 18.1 Sample curves of frame rate functions	285
Figure 18.2 Integrating time control to the game engine.....	287
Figure 19.1 Default Interface and ParaIDE	296
Figure 19.2 ParaEngine components, API and programming language.....	297
Figure 19.3 World Inheritance	298
Figure 19.4 Creating a New World	298
Figure 19.5 Sketch of Leonardo da Vinci.....	299
Figure 19.6 Sample Scene Screenshot 1.....	300

Figure 19.7 Sample Scene Screenshot 2.....	300
Figure 19.8 Terrain exporter in 3dsmax	301
Figure 19.9 Apply Terrain Modifier in ParaIDE	302
Figure 19.10 ParaX Model Exporter	305
Figure 19.11 Mount Bone and Mount Animation	307
Figure 19.12 Mount one character over another.....	308
Figure 19.13 Event handler properties in the property window of default interface.....	309
Figure 19.14 Empty event handlers script file in the editor	310
Figure 19.15 NPC database in ParaIDE	311
Figure 19.16 Follow controller parameters	313