Design and Implementation of Distributed Game Engine

- A Cognitive Simulation Approach

by

Xizhi Li

A thesis submitted to
Zhejiang University
in partial fulfillment of the requirement for
the degree of
Bachelor of Science

Advisor: Prof. Qinming He

Hangzhou, P.R. China, 2005

Copyright Page

I hereby declare that I am the sole author of this thesis.

I further authorize Zhejiang University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature

ParaEngine Copyright Notice

ParaEngine is an unreleased property of the author and can not be modified and/or redistributed without the permission of the author. The code and documentation of *ParaEngine*, as provided in the self reference section of this thesis, is also copyrighted as above. Examples of code, pictures, and text relating to *ParaEngine*, which appear in the thesis's content and appendices, may be reproduced for any purposes. The author is grateful if he can be informed of its usage.

For distribution of other materials from my own reference, please contact me via email

lixizhi@zju.edu.cn or lxz1982@hotmail.com

Abstract

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. This thesis is to exploit this possibility using established computer technologies as well as newly designed ones. The topic of the thesis is on the design and implementation of a distributed computer game engine called ParaEngine. ParaEngine is an experimental game engine framework aiming to bring interactive networked virtual environment to the Internet through game technologies. It has been developed in close conjunction with an actual computer game – Parallel World, a distributed multiplayer Internet 3D game – world and logics on individual computers can be linked together like web pages to form an interactive and highly extensible gaming environment.

Current game engine framework already include solutions for a large number of platform issues, such as real-time 3D visualization, physics simulation, event and script system, path-finding, high-level decision making, networking, etc. However, the computing paradigm behind it is usually constrained to a single platform, where a predefined network topology must be explicitly constructed for cross-platform communications. One of my major research goals is to redesign the computing paradigm to suit the need of highly dynamic networked virtual environment, where intelligent entities are situated and communicate with each other as well as human avatars, and to implement the new paradigm in a full-blown game engine.

In the design of ParaEngine, the overall computing paradigm is compared to the cognitive simulation process, in which simulations, visualizations and interactions are related in a distributed and network transparent manner. The cognitive simulation framework is studied and partially implemented in the game engine by means of the scripting system. Other aspects of ParaEngine are also discussed, such as model rendering, 3D navigation and game AI, etc. However, emphasis has been put on key techniques, as well as how they can be efficiently integrated in an actual game engine.

The organization of the thesis follows the design and architecture of ParaEngine, which is written at a level of detail that allows for researchers and practitioners to design and build the next generation distributed game engines.

Key words: Distributed game engine, networked virtual environment, simulation

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Game engine is an all encompassing subject in computer science. 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 highest level of artificial intelligence and art 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. This thesis is to exploit this possibility using established computer technologies as well as newly designed ones. The topic of the thesis is on the research and implementation of a distributed computer game engine called ParaEngine. ParaEngine is an experimental game engine framework aiming to bring interactive networked virtual environment to the Internet through game technologies. It has been developed in close conjunction with an actual computer game – Parallel World, a distributed multiplayer Internet 3D game. In this game, worlds and logics on individual computers can be linked together like web pages to form an interactive and extensible gaming environment. Please see Figure 1.1 for some screen shots of the distributed game engine that I developed.

Figure 1.1 ParaEngine Screen shots

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

framework of a computer game engine. It is safe to predict that game technology will continue to be a rapidly reforming area in the future.

In this chapter, I will review game engine technologies and related research, and point out a more or less special game engine framework, which I called, Distributed Computer Game Engine. The scope, organization and major contribution of the literature are given afterwards.

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. See also [1] [2] [3] [4] [5] [6] [7] [8].

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 Introduction to 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 Novodex [9] and Havok [10] 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 outsourcing [11].

The continued refinement of game engines has allowed a strong separation between rendering, scripting, artwork, and level design. It is now common (as of 2005), 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 usually used 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; 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, level editor, 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 engines [12] [13].

Table 1.1 shows a quick a quick jot-down list of game engine technologies which are used in my own computer game engine, ParaEngine.

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/texture sorting, terrain, clipping/culling/occlusion, frame buffer post-processing effects, split-screen support, mirrors/reflection, procedural geometry, text rendering/font issues, level of detail, projected textures, …
Resource management	loading, resource lifetimes/garbage collection, streaming resource scheduling, rendering device change management, file access (pack 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 Networked Virtual Environment and Virtual Reality Engine

Prior to game engine, there have been researches in networked virtual environment [1] and the supporting framework called virtual reality engine (VR engine). Some recent trends proposed by its community can be found at the Web3D consortium [2], such as the use of VRML and X3D scripting languages.

1.1.2.1 Research Motivation

My research works on game engine does not originate from observing these activities on virtual reality engine or networked virtual environment. But when I knew them, I liked their ideas very much. Hence, it can now be regarded as a shared motivation between me and people working in this area.

The original motivation for my research on game engine has much to do with my interest in Artificial Intelligence or AI. To me, AI is a very broad subject. Generally speaking, my work in the past is divided into two groups. In group one, I figure out: what is going on in our biological mind; is there any minimum set of high-level rules governing this process; how they can be best approximated by established computer technology; what applications and implications do they have. In group two, I am doing some innovative framework researches regarding the architecture of computing paradigms in the future. I believe that the computing paradigm will be centered on intelligence. Prior to game engine, I have some early research works on the Web Agent Framework [14] (a Multi-Agent Systems), Distributed Human Computer Interface (DHCI) [15], and a network-transparent scripting system called NPL [16]. Almost at the same time, I took an undergraduate game development course at Zhejiang University which brought me to the new realm of game engine research. I found that all my prior works can be integrated in to a new framework, which I called, Distributed Game Engine Framework.

Although there are shared visions between distributed virtual reality engine (which I will explain later) and distributed game engine. I prefer to use the name Distributed Game Engine throughout this literature to emphasize its close relationships with games and personal computer (PC) platforms.

1.1.2.2 Virtual Reality Engine Projects

The following VR engine projects will be reviewed: Dive, VREng, CVE and Croquet. They all support networked virtual environment. So they are also called distributed VR engine.

1.1.2.2.1 DIVE

The SICS Distributed Interactive Virtual Environment (DIVE : http://www.sics.se/dive/dive.html) [17] is an experimental platform for the development of virtual environments, user interfaces and applications based on shared 3D synthetic environments. Dive is especially tuned to multi-user applications, where several networked participants interact over an internet.

Dive is based on a peer-to-peer approach with no centralized server, where peers communicate by reliable and non-reliable multicast, based on IP multicast. Conceptually, the shared state can be seen as a memory shared over a network where a set of processes interact by making concurrent accesses to the memory.

Consistency and concurrency control of common data (objects) is achieved by active replication and reliable multicast protocols. That is, objects are replicated at several nodes where the replica is kept consistent by being continuously updated. Update messages are sent by multicast so that all nodes perform the same sequence of updates.

The peer-to-peer approach without a centralized server means that as long as any peer is active within a world, the world along with its objects remains "alive". Since objects are fully replicated (not approximated) at other nodes, they are independent of any one process and can exist independently of the creator.

The dynamic behavior of objects may be described by interpretative scripts in Dive/Tcl that can be evaluated on any node where the object is replicated. A script is typically triggered by events in the system, such as user interaction signals, timers, collisions, etc.

Users navigate in 3D space and see, meet and collaborate with other users and applications in the environment. A participant in a Dive world is called an actor, and is either a human user or an automated application process. An actor is represented by a "body-icon" (or avatar), to facilitate the recognition and awareness of ongoing activities. The body-icon may be used as a template on which the actor's input devices are graphically modeled in 3D space.

A user `sees' a world through a rendering application called a visualizer (the default is currently called Vishnu). The visualizer renders a scene from the viewpoint of the actor's eye. Changing the position of the eye, or changing the "eye" to another object will change the viewpoint. A visualizer can be set up to accommodate a wide range of I/O devices such as an HMD, wands, data gloves, etc. Further, it reads the user's input devices and maps the physical actions taken by the user to logical actions in the Dive system. This includes navigation in 3D space, clicking on objects and grabbing objects etc.

In a typical Dive world, a number of actors leave and enter worlds dynamically. Additionally, any number of application process (applications) exists within a world. Such applications typically build their user interfaces by creating and introducing necessary graphical objects. Thereafter, they "listen" to events in the world, so that when an event occurs, the application reacts according to some control logic.

1.1.2.2.2 VREng

VREng (http://vreng.enst.fr) is an Interactive and Distributed 3D Application allowing navigation in Virtual Environments connected over the Internet using Unicast or Multicast if available. Worlds and their objects are described in a XML format.

VREng is a Web3D [2] which allows its users to navigate in Virtual Worlds like rooms, campus, museums, workshops, landscapes, networks, machines,... Visitors may interact with each other through their avatars. They may also communicate by exchanging short textual messages (Chat), audio and/or video channels, shared white-boards and interact with objects in the 3D environment like Web panels, virtual workstations, documentation on-line, MP3/Midi music, MPEG audio/video clips, MPEG4 animations, and remote applications and servers. Moreover, the user can put into worlds all kind of electronic documents and publish them to other participants on a message board keeping persistency.

VREng gives the opportunity to experiment with a more attractive and pleasant way accessing to multimedia data usually located in Web sites.

1.1.2.2.3 CVE

CRG Virtual Environment (CVE) [18] is also referred to as MASSIVE-2 in some papers. CVE is a distributed multi-user virtual reality system, current features of which include: networking based on IP multicasting; support for the new extended spatial model of interaction, including third parties, regions and abstractions; multiple users communicating via a combination of 3D graphics, real-time packet audio and text; extensible object oriented (class-based) developers API.

1.1.2.2.4 CROQUET

Croquet (http://www.opencroquet.org/) is a combination of open source computer software and network architecture that supports deep collaboration and resource sharing among large numbers of users. Such collaboration is carried out within the context of a large-scale distributed information system. The software and architecture define a framework for delivering a scalable, persistent, and extensible interface to network delivered resources.

The integrated 2D and 3D Croquet interface allows for co-creativity, knowledge sharing, and deep social presence among large numbers of people. Within Croquet's 3D wide-area environments, participants enjoy synchronous tele-presence with one another. Moreover, users enjoy secure, shared access to Internet and other network-deliverable information resources, as well as the ability to design complex spaces individually or while working with others. Every visualization and simulation within Croquet is a collaborative object, as Croquet is fully modifiable at all times.

1.1.3 VR Engine vs. Game Engine

If we look at the development of the game engines and compare that to the development of the VR-engines there is one major difference. The game engine has and will be created for mainstream personal computers and console platforms, where as the VR engine up till now has been created for a high end system like a SGI super computer. Until right after the turn of the century, the high-end VR-systems outperformed the game systems by being capable of handling several orders of magnitude more polygons, textures and fill rates. The VR input

system was and still is, quite a bit more advanced than the average home computer. 3D tracking devices and advanced audio video input are very expensive and fragile, and probably will be for some time.

The gaming market is always craving the best visual effects and computer art. This has caused the gaming industry to develop game engines that gets the very maximum out of the available hardware. The competition has been tight, and optimization and quality has been vital for the sales. Huge amount of development money have been put into the development of the various game engines available today.

The VR industry being focused on high-end systems have had a much smaller market. In many ways a more lucrative marked. When the computer system cost more than a million dollars, the software was possible to sell for tens of thousands of dollars, and there was hardly any competition.

1.1.4 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 future virtual environment. Game development is both technology and art. The 2004 game report in [19] provides with timely analysis and actionable insights into emerging game technologies and their potential impacts on existing and new technical education curricula.

1.2 Introduction to Distributed Computer Game Engine

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.

More than just hyperlinks and content transfer, in distributed game world, interactions among entities will be very intensive and extensive, such as several characters exchanging messages at real time; game world logic will be distributed, with each node being a potential server, and also more dynamic, with different nodes forming temporary or long lasting relationships.

1.2.1 Definition of Distributed 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. This is the simple definition I used for this literature.

The definition left out many other aspects in a computer game engine or a virtual reality engine, so that any game/VR engine may be well referred to as a distributed game engine once it acquires such functionalities. However, most of present day commercial or open source games engines have not been in its category, because games created on these engines are either standalone or targeting a fixed server topology.

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; yet its supporting framework may still be called distributed game engine.

1.2.2 Research Problems and Difficulties

I have designed the entire game engine framework from the very beginning to meet the requirement of distributed game engines. Modern computer game engine has evolved to become a complete suite of virtual world constructing tools and runtime environment. The latter is usually a tightly integrated framework of 3D rendering engine, scripting engine, physics simulation and networking. The balance of efficiency and flexibility is the primary issue that is weighed constantly in these many different places in an engine designer's mind. It is usually such compromises drawn by the designer that determined the characteristics of the engine and hence the type of games that could be composed by it.

This section identifies the minimum features that a game engine must support in order to host distributed game worlds. We will see that some of the requirements are rather contradictory. Some commercial game engines may support them as different configurations; i.e. at any one time, only one configuration is valid. However, in distributed game engine, all requirements must be satisfied in a single configuration throughout the framework.

List of distributed game engine requirements:

- **Network transparency.** Game content and logics need to be exposed as named Internet assets. There will be no explicit network modules in the game engine; instead logics spanning the network need to be written in a network transparent manner (i.e. without the knowledge of the actual network where the game logics will be deployed).

- **Scripting.** Everything in the game world should be scriptable, such as resource management, map loading, active objects, 2D GUI, camera and object control, etc. This is because game world logics must be expressed in script (text) form in order to be extensible and transferable on the Internet.

- **Real coordinates.** All scene objects are specified in real coordinate system in the game engine. This ensures that there is no boarder or tile limit to restrict the positioning of game world objects. This may raise problems with collision detection and path-finding in the game engine.

- **Absolute positioning.** Although the scene graph is hierarchically structured (e.g. in a quad tree), absolute values should be used wherever possible to record object positions in the game engine. It is required to move and reposition objects in a dynamic game world. Relative positioning will have some trouble with current simulation algorithms, since we will concurrently simulate a large set of objects which might be geographically far from each other.

- **Dynamic scene loading.** The 3D world can be dynamically loaded and modified in very fine grains and real-time efficiency should be achieved. This is because the focus of simulation will be constantly changing, e.g. a player flies through a long distance in the game world, causing simulation data to be frequently loaded and unloaded. The user, however, must enjoy a continuous visual experience of the changing environment.

- **Wide area simulation.** There is usually no central simulation scheme in the game engine. In other words, we can not assume that game world logics are only activated near the current player or camera position. Instead, the game engine must be able to concurrently simulate game world logics even at places that are far away from the current camera position as long as their activities are potentially related to the state of the current game world. The termination

of a simulation process may optionally depend on time out, memory capacity or explicit commands.

- **Physics and path-finding.** Physics and path-finding are build-in routines of the game engine. They shall be valid wherever the game world logics are still active. Since these routines both eat lots of memory and CPU resources, they must be carefully scheduled for the game engine to be interactive. Sometimes, we have to sacrifice some accuracy for efficiency.

- **Error tolerance.** Physical simulation, human carelessness and network transfer delays or misorderings will all bring errors to the game world, such as a player is stuck in a wall or can not get out of a closed area. The game engine should be able to recover from such situations automatically or on demand.

- **Garbage collection.** Since there is usually no navigation restriction with distributed games, it will not be long before the engine is fed with too many 2D/3D resources and their instances. An automatic garbage collection mechanism must exist so that unused text, textures, sounds, geometries and animations, etc. can be removed from the memory without human intervention.

- **Efficiency.** The amount of computation must be roughly proportional to the number of concurrently active objects, rather than to the geographical size of the map (game world). Hence, the algorithms for rendering and simulation should be modified to quickly find the most relevant data from a large set of input.

1.2.3 Proposed Approaches

The overall architectural design of the game engine is improvised by the cognition process of the human brain, which is reified in the scripting system of the game engine. I will dedicate an entire chapter (Chapter 2) to its explanation and implementation. A similar approach will be used in autonomous character animation in the game engine, which is still an ongoing work.

Other parts of the game engine use modified or tailored versions of established technologies which are common in modern game engines. However, their combinations can be extremely versatile and there are small tricks everywhere. It is safe to say that no two game engines in the world even look like each other.

1.3 Contributions and Results

The major contribution of this literature is a clarified view of distributed game engine platform as well as a good-to-run implementation to begin with. In addition, I will propose my own solution to the following issues:

- Simultaneous visualization and simulation of distributed game world content and logic
- Scripting language runtimes in distributed virtual environment
- Time management in distributed game worlds
- Mixing physics simulation on distributed game worlds
- Autonomous character animation generation

This thesis, 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. I hope that the concept of distributed game engine in this literature can be

accepted and more people will continue this concrete work with their own improved visions for future networked virtual environment.

The major research result is a distributed computer game engine, called ParaEngine. ParaEngine is an experimental distributed game engine framework that I have designed and implemented since early 2004. It is not based on any commercial or open source game engines. I have designed it from the very beginning to meet my specific requirements. It is also developed in close conjunction with an actual computer game – Parallel World, a distributed multiplayer Internet 3D game – world and logics on individual computers can be linked together like web pages to form an interactive and highly extensible gaming environment. I also host a website on my personal web. You are welcome to it for the latest news and download.

ParaEngine URL: http://www.lixizhi.net/paraworld/web/index.htm

The framework presented in this literature is based on ParaEngine, unless I explicitly declare that it is still under development. Therefore, the architecture and approaches used in this paper are guaranteed to function well with current computer hardware.

1.4 Organization of the Thesis

Chapter 1 introduces game related technologies and the background and motivation of distributed computer game engine. Chapter 2 proposes the major theoretical framework for the distributed game engine.

Beginning from Chapter 3, the organization of the thesis follows the design and architecture of a specific game engine framework called ParaEngine. All aspects of the game engine framework will be discussed. However, emphasis has been put on key techniques, as well as newly proposed frameworks, such as the network-transparent scripting system, the time management system, etc. Chapter 3 provides an overview of the entire game engine architecture. Chapter 4 is a big chapter, explaining the data presentation of 3D game world and the rendering pipeline and techniques. Chapter 5 focuses on all kinds of simulations carried out by a game engine. Chapter 6 deals with navigation in the 3D world. Chapter 7 shows the implementation of the scripting system and how it can be used to compose the distributed game world in a network transparent manner.

Chapter 8 is an additional chapter about AI in games. AI technique is specific to games, and there are both simple and complex ways to create intelligent behaviors for characters in a game. Chapter 9 discusses time management or frame rate control in the game engine. The final Chapter 10 concludes the paper with an outlook of distributed game engine and my future plans.

Two categories of references are provided. One is formal bibliography; the other is ParaEngine's design document. With these references, I try to direct readers to free resources on the Internet for further information on each topic.

Chapter 2
The Cognitive Simulation Framework

A computer game engine mainly involves graphics rendering, content building, and simulation. 3D world rendering and modeling techniques are largely the same for almost all kinds of 3D game engines. It is the simulation framework that distinguishes one game engine from another.

A simulation system has input, output, memory and a computing mechanism. The high level view of a game engine can be most appropriately described in terms of a simulation system. For example, it reads user input from key boards and mouse movement; maintains a memory of the entire game world states; simulates the game world for a small time step according to some predefined rules; and outputs result in cutting-edge multimedia form to the user. Big simulation systems are comprised of smaller ones and may interact with its peer systems.

In this chapter, I introduce the simulation framework adopted by ParaEngine. This framework will be applied to the scripting system in the game engine. Since it is still my ongoing research, I will only present a high level picture here.

2.1 An Analog to Human Brain

Motivation of the distributed simulation framework in the game engine came from the introspection that human beings are capable of generating first-person and third-person simulations in its brain. This capability leads us to one of the famous hypothesis concerning the human brain, which is called the Simulation-Theory (ST). The following two examples illustrate some of its basic ideas. Example one: suppose you are now reaching out your hand for a cup of coffee, why would you do this action? The hypothesis explains that you reach out your hand because you have simulated this action in your brain slightly before you perform it. And you reach out your hand instead of using it to move the mouse or stroke the keyboard (which might probably be the next actions if you are using a computer at the same time), possibly because your attention is then on the feeling of thirst and the cup of coffee. This does not mean that your brain was not simulating moving the mouse or stroking the keyboard at that time. Instead, these actions might also being simulated then, but they were neither performed nor emerged in the conscious brain because they were not magnified due to a selection process of the attention. Example two: suppose you are driving a car when there appear some people about to cross the road, how do you decide whether to brake your car or not. The hypothesis tells that your brain is continuously simulating the observed motion of the people ahead of time so that the attention of danger will not be signaled unless the simulation leads to a similar dangerous situation stored in the mind. And because simulation takes place before it actually happens, the redistribution of attention during simulation decides whether you brake or not. I.e. if danger is signaled, the shifted attention will magnify (select) the brake simulation and braking will be performed.

For years, researchers have suspected that the binding task (mind and brain) is accomplished by nerve cells in distinct areas of the brain communicating between themselves by oscillating in phase (40 hertz) -- like two different chorus lines kicking to the same beat even though they're dancing in different theatres. These oscillations have been detected in

everything from the olfactory bulb of rabbits to the visual cortex of cats and even conscious humans. IBM, Birmingham and Saint Mary's researchers believe they have explained not only how the oscillations come about but also how the oscillatory rhythm is communicated from one area of the brain to another. These two findings are critical to understanding how the complex electric signals of large numbers of nerve cells generate awareness and perhaps even consciousness.

This coherency or synchronization among different groups of nerve cells is a sign of concurrent simulations in the brain. According to the ST theory, by performing simulations in a similar manner, it is possible to generate new motions for an animation system. In our proposed motion generation framework (see Chapter 8.4), a learning and simulation algorithm is used to generate the motion for the characters. We first combine the animation variables and all the related environment variables which form the configuration space called DIM (dimension) space. Each variable or DIM in the configuration space is a function of time *f (t)*, *t<=Tc*. Any subset of *these* DIMs and a segment of time [*T1*, *T2*] can be regarded as a simulation of these synchronized DIMs of length (*T2*-*T1*). An attention mechanism is used to keep track of and select the most relevant simulation(s) for each DIM. The value of *f (T+ ⊿T)* for every DIM is computed from its selected simulations unless overridden by an external user (supervisor or environmental inputs). Details will be given in Chapter 8.4.

2.2 Motivation from the Simulation-Theory

The Simulation-Theory (ST) is originally a theory about how the human brain generates visual imagery. A recent description of this theory can be found in papers by Hesslow [20]. I will further develop this theory or hypothesis to make it easy to understand from a computer point of view.

ST states that human imagination and visual/auditory perceptions are in essence the same thing in our conscious mind, and that they are both the input and output of the unconscious mind which does the work of recognition, memorization and deduction. The cycle of imagination and the subconscious forms mostly a closed loop when we are asleep, and a biased loop (by what we perceive) when we are awake. See Figure 2.1.

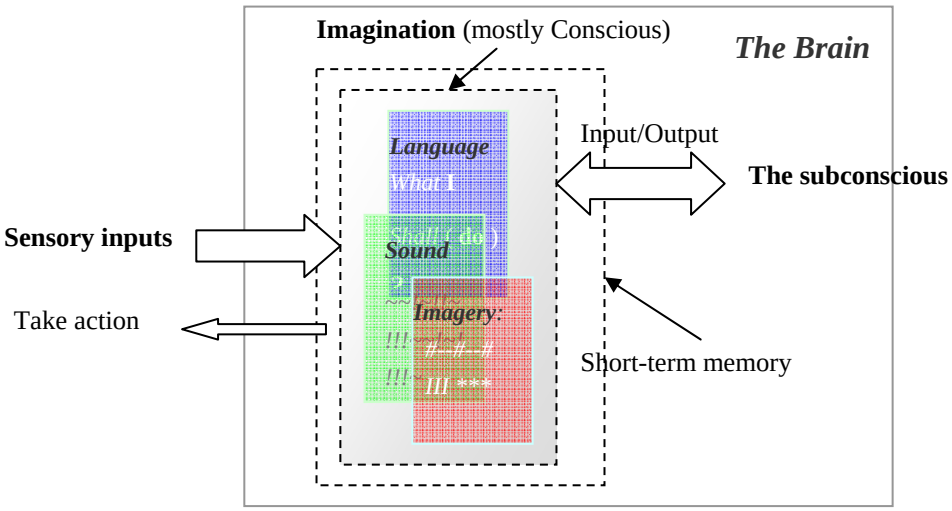

Figure 2.1 Human Brain: The Imagery-subconscious loop

Metaphorically speaking, the Imagination can be thought of as a multimedia, virtual reality "theatre" [9], where stories about the body and the self are played out. These stories,

- are influenced by the present situation according to perception,
- elicit the subconscious activities accordingly,
- and thereby influence the decisions taken by action.

The following parts are the key ingredients in the ST theory. They are "dimensions of simulation", "concurrent simulation", "imagery-subconscious loop" and "attention and memory".

(1) **dimensions of simulation.** Any real world situation can be mathematically dissected into infinite number of functions $f_n(t)$ with each evolving over the same time variable *t*. Given *N* samplings of these functions, we obtain an *N*-dimensional simulation of a real world situation, i.e. $\{ f_n(t) \mid n = 1, 2, ..., N; t \in [T_0, +\infty] \}$. Generally speaking, the similarity between the simulated situation and the real world situation increases as the number of dimension *N* increases. So long as the number of dimensions is high enough, the two very different systems (i.e. the simulated system and real world system) are analogous to each other. We will propose later that the number of concurrent simulations in the human brain is not one, but many; and they are simulated with varying degrees of dimensions or similarities to the real world. Simulation with the highest dimensions is most analogous to the real world situation and becomes our conscious or internal perceptions. The brain mechanism controlling the selection of simulation dimensions is called "Attention".

(2) **concurrent simulation.** At any given time, our brain is simulating thousands of visual/auditory/etc. events concurrently. A dimension f(t) may be reused for different simulations at different times. Concurrent simulations are of high interest or relevancy to the current state of the brain. But only a selected few are enlarged (i.e. their number of dimensions is increased) to be perceptible in our inner world (consciousness).

(3) **imagery-subconscious loop.** This part has been described in many other literatures [20] [21]. It just states that simulation in the human brain can evolve on its own, without interacting with the real world. This is achieved by feeding the result of a simulation to the input of itself, hence forming a loop between imagery and subconscious in the brain.

(4) **attention and memory.** Attention selects only a limited mount of imagery at any given time, despite there might be millions of other stories that are being played (simulated) in the mind at the same time. It is the constant selection of our attention that constitutes what we perceive as a continuous consciousness. Attention replays a previously unmagnified simulation or memory clip in the same sequential order as it was generated some time ago, therefore reinforced it in the memory; it signifies the importance of such imagery by bringing it to our internal perceptions, which in turn, makes it easier to affect subsequent imagery generation and selection of attention.

2.3 Learning and Total Feedbacks

The simulation theory also suggests ways of acquiring new knowledge from its environment. However, unlike other learning system, one of its assumptions is that there are already

patterns in its input stream. In traditional learning systems, patterns are extracted from statistics of inputs and feedbacks, which means that it usually requires many supervised feedbacks in order to learn a simple case. However, in a simulation based learning system, feedbacks are considered to be already contained in its input stream. The input stream is treated as ordered and rational multidimensional data. Hence, the new learning system requires little or no supervision to learn a case. This is also true with the human brain, which rarely learns a pattern through statistics of many inputs and supervised feedbacks. Instead, it learns by example (i.e. simulation). I call feedbacks in simulation based learning system "*total feedbacks"*, in order to distinguish it from feedbacks in other learning framework. In other words, all the dimensions of the input stream are treated both as input and feedback without being explicitly appointed which is which. *Total feedback* is an important concept which generalizes the characteristics of many intelligence simulations in computer game engine.

2.4 Argument and Conclusion

Many people may argue the necessity to draw an analog to the human brain, where the framework may well be described in mundane terms. The following paragraph is my short answer to the argument. However, if you are the reader who still insists on a more technical explanation to the simulation framework, I ensure you that, in the rest of the literature, you will see either mathematics or concrete implementations of the proposed simulation framework, but no more descriptions of the human cognition, although I liked this chapter very much.

The following is my answer to the "necessity argument".

- If there is really an analog between the simulation framework and the human brain, why avoid using it?
- Why fear to say so, when a small contribution may be useful to AI researches? In recent years, important techniques developed by AI researchers have been bound to their most relevant application domains, and there is no more theory or technique which solely belongs to AI. I think this is due largely to the fear of attributing bits of new contributions to a field of study which have failed so many times in the past. Yet, it has been very successful in its many application domains as techniques.
- For a long time, the study of artificial intelligence does not have a unifying framework to take advantage of its many developed theories and techniques. This is why other disciplines have clipped AI in to small pieces and possessed the share useful in their applications. The proposed simulation system (maybe game engine) is likely to bring back what is taken from AI researches.

Chapter 3
Distributed Game Engine Framework

3.1 ParaEngine Framework Overview

The core of ParaEngine is comprised of the following four components.

- 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.
- 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 (scene graph), with a root node on top called scene root. The rendering pipeline is built directly in the scene manager. Some physics calculation (such as local animation and particle motion) is also performed by it.
- Environment Simulator: At every cycle, the environment simulator advances the game state by a small interval. It takes input from the user, AI modules, and the networking scripts, 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 script systems, etc.
- The Scripting system: The high level game logics are all expressed in script files which may be distributed on a computer network. For example, game content, graphic user interface, character logics, and game state synchronization, can all be written and controlled entirely from script files.

Figure 3.1 shows an architecture overview of the major engine components.

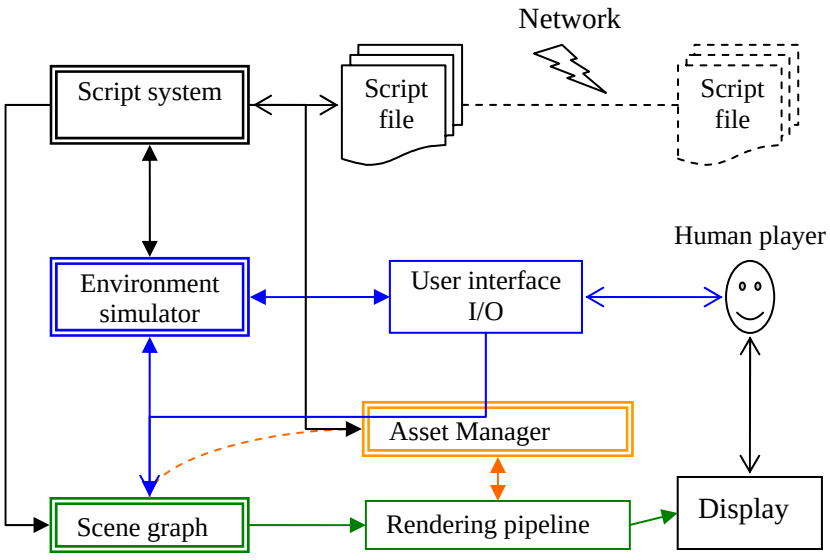

Figure 3.1 Overview of the major engine components.

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 with multi-threading, message driven and event based system. The infinite loop in a computer game engine is called game loop [22], 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 3.1. It has been extended to include more details of the engine framework.

Table 3.1 Game loop

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

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 runs 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 engine.

}

}

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, script language runtime, network daemons and physics engine. In most cases, Windows message handler and script language runtime are running in the same thread as the game loop. However, since no knowledge is known about the activation rate of the Windows message handler, it is treated similarly with multithreaded functions with the exemption of handling exclusive data writing. Physics engine such as Novodex [9] is multithreaded; while most other physics engines are still single threaded. Network daemons are inevitably multithreaded. Some thread-safety issues should be properly dealt with in multithreaded environment. A common method is to duplicate public data which can be seen by both the working thread and the game loop. A benefit of this method is that the game loop can always access data in a uniform manner. Based on the above game loop, we will review several related techniques in the following sub sections.

3.3 Timing and Engine Modules

In ParaEngine, several global timers are used to synchronize engine modules that need to be timed. Figure 3.2 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.

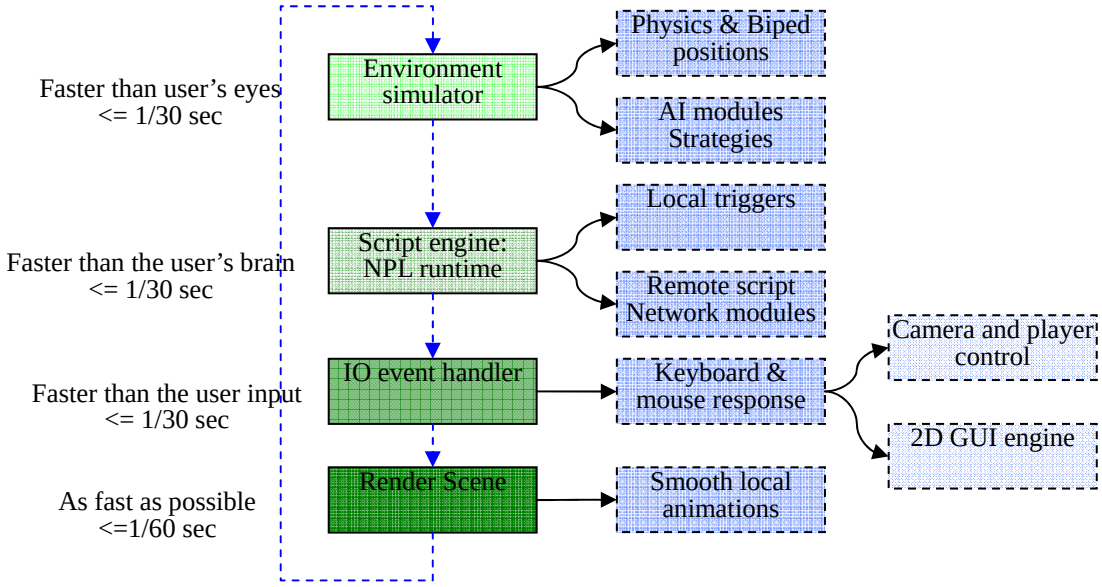

Figure 3.2 Timing and I/O in ParaEngine

3.4 Game World Logics Division

One of the major tasks of a computer game engine is to offer language and tools for describing Game World Logic in an engine digestible format. Usually the logic of the entire game world can be further partitioned into three subcategories of programming: (1) script programming: it is most flexible and can be distributed in many files. Our objective in designing ParaEngine is to let Scripting do as much as possible. (2) C++ programming: it

extends basic functionalities already provided in the engine core, such as some computational intensive AI strategies. (3) Engine programming: It is fixed with the release of a game engine. Common functions such as physics and path-finding routines are in this category.

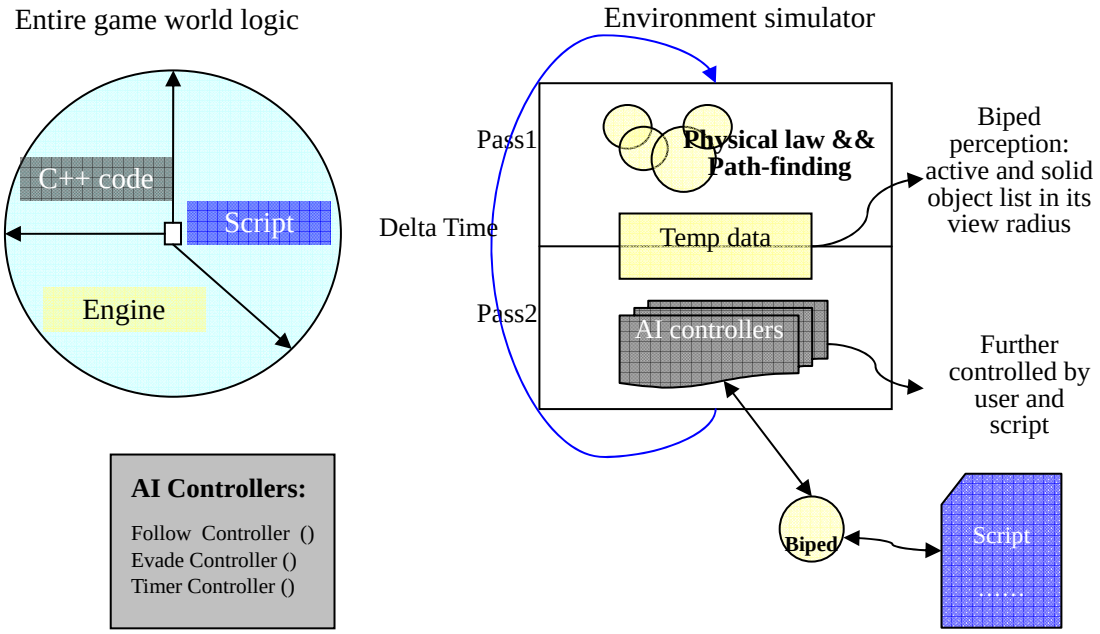

Figure 3.3 Game world logics in ParaEngine

In Figure 3.3, AI controllers are C++ code based AI modules that should not be confused with Script-based AI. AI controller is the best choice whenever performance is most critical. They can be assigned to Biped Scene Object. The association of biped object and AI controller is entirely arbitrary and reassignment is also allowed during game play through scripting. For example, in the game demo Parallel World, a “Follow” Controller and “Evade” Controller were written in C++ code, so that when assigning a NPC creature to both of them, it will have the ability to follow automatically a moving target (avoiding all obstacles in its view perception) as well as evade a target according to its unit type (melee or ranged). Both the AI controller assignment and controller-control are available as host APIs in the script language used by the game. A useful conclusion of this section is that although we hope to build everything from scripts, there still exist some places where current scripting technology is not eligible or suitable to use. In other words, languages such as (VRML + Java) might alone be capable of static and/or interactive 3D information visualization on the web, it is not sufficient, though, to implement all game world logics required by a modern Internet computer game in an efficient way. This is one reason why hard-coded logics are still being used in ParaEngine and many other computer game engines.

3.5 Conclusion

Game-engine practitioners have used scripting technology to add soft computing capabilities to a variety of their engine modules; so that commercially released games will still enjoy a certain degree of online-reconfiguration. Unfortunately, there has been no unified approach of doing it. Instead, most game engines explicitly implement a network module which usually relies on Clients/Server (CS) architecture and a single session. A central database is used to

hold all the status of its game entities and all events and triggers in the virtual game world reside only on the local computer. ParaEngine overcomes these limitations and provides further flexibilities by means of using a network transparent scripting system in constructing game world logics.

We will cover the details of ParaEngine in the following Chapters.

Chapter 4
Modeling and Drawing the 3D world

4.1 Resource Management

Managed game resources are all derived from a base class called AssetEntity. It maintains a reference count for each asset instance. Figure 4.1 shows a list of entity types currently supported by ParaEngine.

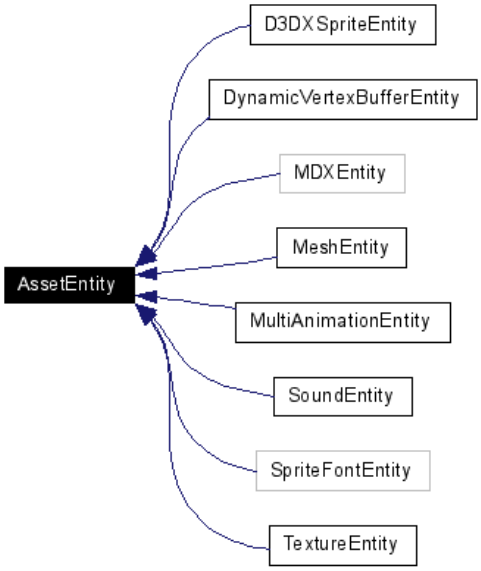

Figure 4.1 Asset entities in ParaEngine

Asset entity can be created at any time during the game. In most cases, game assets are created at the beginning of each level. Asset entity itself must be associated with scene object to take effect. And scene object must be bind to asset entity in order to be rendered. The same asset can be shared among many scene objects.

Asset entities are hashed using its resource file names. 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. Only one short name can be associated with an asset, if multiple names are assigned to the name entity, the latest assigned name will override previous ones. In ParaEngine, scene node stores asset entities as object pointers. Entity is by default lazily initialized, unless they are called to be initialized. In other words, an asset is only initialized when it is used for the first time in game engine pipeline.

4.1.1 Virtual File Management

All resource files are exposed through the CParaFile interface. It can be a real file on the disk or a virtual file in a large zip file or a file in different search paths. Virtual file management makes resource management easy by exposing a unified programming interface for all kinds of files used in the game engine, regardless of the actual location of the files.

4.2 Scene Management

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. The scene object is usually not concerned with how it is finally rendered. Instead, each of them is associated with one or several asset entities. These asset entities are usually textures, mesh and animation data that would contains the actual graphical data for rendering, as well as methods of drawing the objects.

- **Asset entity.** Asset entity can be created at any time during the game. In most cases game assets are batch loaded at the beginning of a new simulation by a script. Please refer to Chapter 4.1.
- **Scene objects.** The scene objects are organized in a tree hierarchy, which is optimized for geographical searching of individual objects.

Game scene objects are usually organized in a scene graph [23] [24]. A scene graph is a tree where the nodes are scene objects arranged in some sort of hierarchy. These nodes may be physical objects, or simple 'abstract' objects. For example, we can organize objects spatially in a scene graph, such as in a quad tree and octree, so that it could be used for clipping and occlusion of unseen objects.

In ParaEngine, an automatically generated terrain tile structure is used to organize scene objects in a spatial quad tree. The reason to use a quad tree is that in a big and extensible 3D world, most objects are located in a large flat plane. Quad tree is 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 top level scene organization structures. A specific scene object may organize its internal objects in whatever structures it likes, such as Octree, BSP tree, etc.

Figure 4.2 shows the quad tree terrain tile organization. The scene graph 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 biped object is attached to the Terrain Tile in which it is active, instead of to the tile that best contains it. We know that when the rendering pipeline transverses the scene graph tree, only objects near the camera are visited; however, when the simulation pipeline transverses the tree, all active objects must be covered regardless whether they are near the camera or not. Therefore by moving active objects up near the tree root and using absolute values to store their positions in the scene, it ensures that they can get simulated in simulation pipeline. We will cover the details of simulation (collision detection and response, path-finding, etc) in Chapter 5.

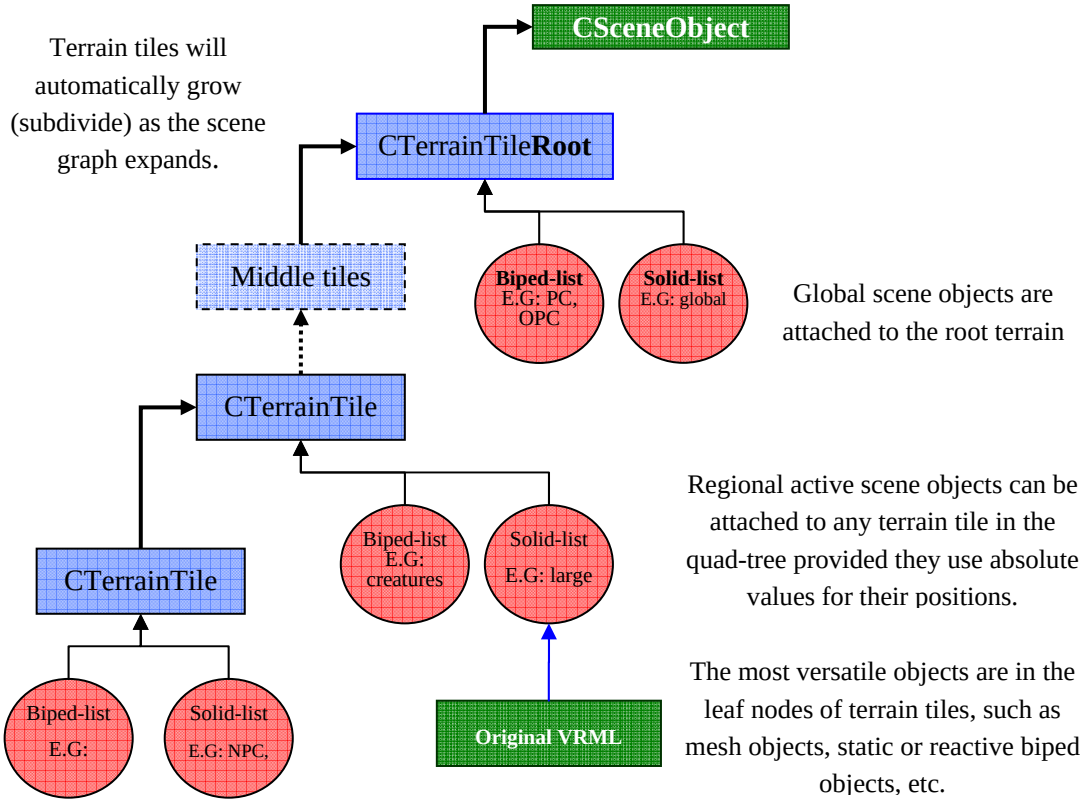

Figure 4.2 Scene graph tree

4.2.1 The Scene Root Object

The Scene Root Object (CSceneObject) is the single most 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 camera, 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, 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 4.3 shows the Collaboration diagram of the class.

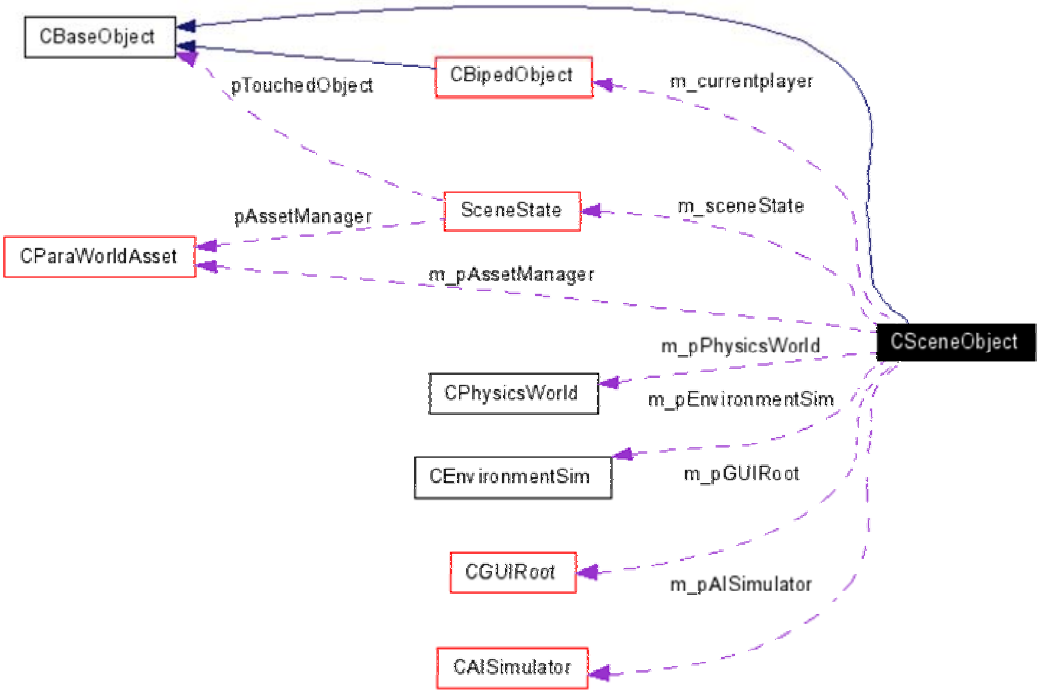

Figure 4.3 The Collaboration diagram for the root scene object

4.2.2 Dynamic Scene Loading and Unloading

A group of scene objects can be managed as a single entity through the managed loader interface. The scripting system can take advantage of this feature. For example, a city of houses can be grouped by one loader; whereas a house with internal decorations can be grouped in another. Please refer to the NPL scripting references (ParaEngine, 2.4) for more information.

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 are attached to the scene graph, the ownership of these objects transferred to the scene manager, otherwise the object ownership is the loader object. The owner of an object is responsible to clean up the object when it is no longer needed. The behavior of a managed loader is given below:

- The child objects of a managed loader will be automatically loaded and unloaded as a single entity.
- Generally only static objects are attached to a managed loader.
- Different managed loaders in a ParaScene must have different names.
- If one creates a manager loader with the same name several times, the same managed loader will be returned.
- The bounding box of a managed loader will be automatically calculated as new child objects are attached to it.

- The current loader algorithm will linearly transverse all managed loaders in the scene to decide which scene objects to load or unload. Although the routine is executed when the CPU is free, it is good practice to keep the total number of managed loaders in a single scene low. I think a couple of thousand loaders will be fine for current hardware.
- It is good practice to use managed loaders to group all static scene objects that appears in a scene. Because, it will keep the scene graph to a moderate size automatically and accelerate physics calculation, etc.
- It is good practice to put all objects which are physically close to each other in a single managed loader.

4.3 Static Scene object Presentation

This section shows the implementations for various static scene objects in ParaEngine. A static scene object does not change its global position once it has been attached to the scene. However, static scene objects may contain local animations, such as undulating waves, moving clouds, swinging lights, etc.

4.3.1 Sky and Fog

In ParaEngine, there are several types of sky. The sky object will automatically adapt to the current fog parameters in the scene manager. In current version, it supports static mesh based sky. See below.

4.3.1.1 Static mesh based Sky

This approach is very simple. It uses the linear fog function already supported by most GPU, hence, no vertex or pixel shaders are used.

It first renders the sky box/plane/dome or whatever as an ordinary static mesh object. Sky mesh is always drawn before rendering any other object in the scene. This is because sky mesh rendering will turn off Z-buffer temporarily. Note that the mesh is not infinitely large (should be smaller than the camera frustum), and it just gives the fault illusion of sky in the infinite distance, provided that the sky mesh center is the camera eye position. Then we retrieve the fog color from the scene manager and blend the fog color to the static background. Currently, triangular strips are used to build the four sides for the bounding box, the vertex color at its base has alpha 1.0 which is the fog color; and 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 plane. 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 a distant object with the sky infinity. In case, a reviewer is looking downward from a mountain, the center point of the triangular fan has the exact color as the fog. Figure 4.4 shows the implementation.

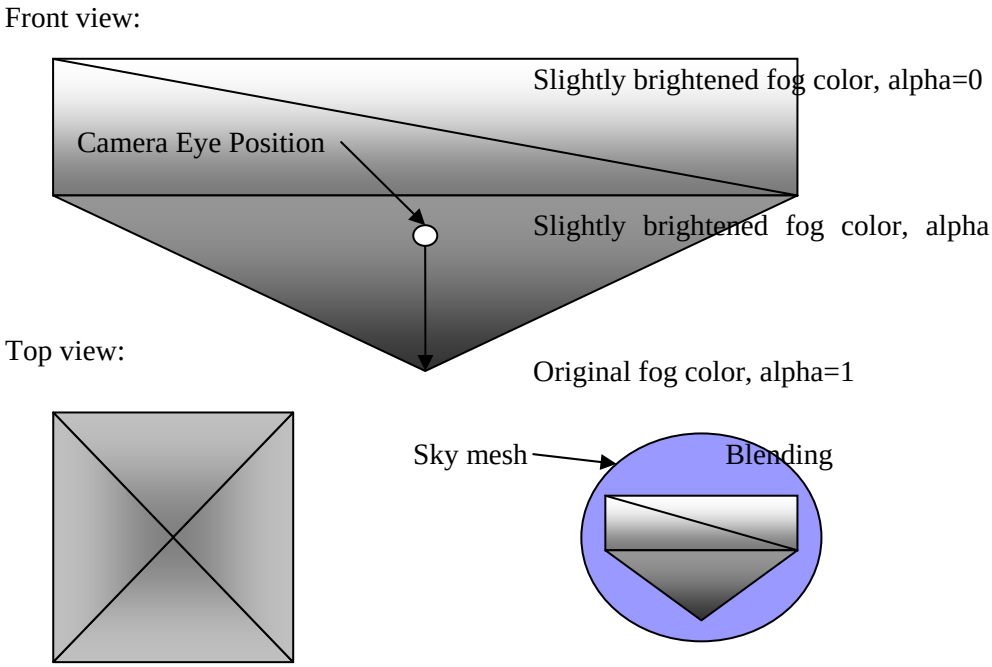

Figure 4.4 Fog color blending

The mesh object whose vertices are colored as shown in the figure is used to alpha blend with the background static sky mesh. Figure 4.5 shows the implementation result. We can observe how textures on the sky mesh, the fog, the terrain and other 3D objects coexist in a single scene. In the next section, I will describe how the object level culling in the game engine pipeline coexists with the fog.

Figure 4.5 Fog implementation in ParaEngine

4.3.2 Fog and Object Level Culling

Object level culling is a technique to remove an entire 3D object at an early stage of the rendering pipeline. In ParaEngine, it is performed in two steps. The first is called rough object level culling, in which the bounding sphere of the object is tested against a view sphere. If the

two spheres intersect, then a second more precise test is performed to see if the bounding sphere or bounding box of the object is inside the view frustum.

In the first "rough object level culling" step, the following algorithm is used to decide the view radius according to the size of the object, the distance to the view center and the current fog parameters. The algorithm is parameterized by the following variables.

- m_dwObjCullingMethod: the current object level method used. The default value is CENTER_ON_CAMERA|VIEW_RADIUS_AUTO.
- m_fCullingAngle : We will allow an object of height fNear*tan(m_fCullingAngle) to pop out from fNear distance from the camera eye. The default value is 5 degrees or D3DX_PI/36
- m_fMinPopUpDistance: All objects must be drawn when they are within this radius. Default value is 15 meters.

m_dwObjCullingMethod is a bit wise field for object level culling parameters:

- CENTER_ON_CAMERA: the view center is always on camera eye position
- CENTER_ON_PLAYER: the view center is on current player. If no player selected. it is on the current camera eye position
- VIEW_RADIUS_FOG: Use fog far plane distance as the view radius.
- VIEW_RADIUS_AUTO: Automatically adjust the view radius according to the fog near and far plane as well as the size of the object.
- VIEW_RADIUS_FRUSTUM: Objects that intersect with the view frustum are all drawn. This is the most time consuming one. The center flag is neglected. The only information to provide the radius and the center is from the current camera.

The following formula is used to decide the view radius in VIEW_RADIUS_AUTO mode. 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. The view radius ViewRadius is given by the formula:

ViewRadius(R) = max((fFar-Pow2(fNear*tanf(m_fCullingAngle))*density*(fFar-fNear)/(R*R)), fNear);

In concise form, v(R) = f – k / (R * R), where f, k are some pre-calculated values. Figure 4.6 shows the curve of v(R).

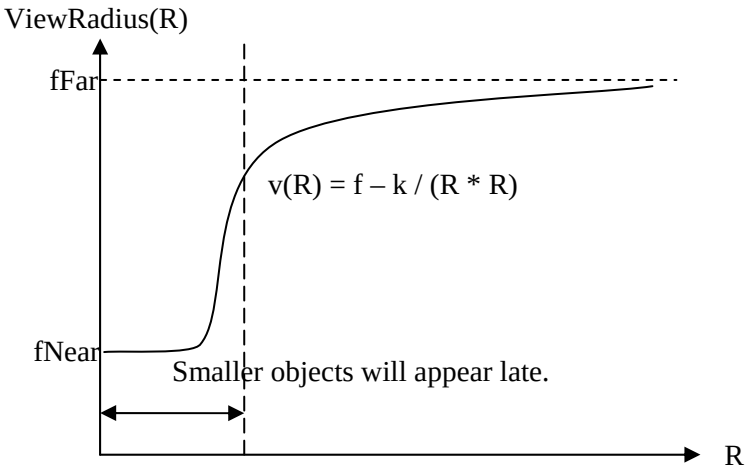

Figure 4.6 Object Level Culling: View Radius Function

The formula ensures that the pixel changes of any newly popped out object are a roughly a constant. Small objects will only be drawn when they are very close to the near fog plane, whereas large object will be drawn as soon as they are within the far fog plane.

4.3.3 The Terrain Engine

4.3.3.1 Introduction

The Global Landscape Terrain engine used in ParaEngine supports infinitely large terrain rendering and fast ray tracing. The core ROAM algorithm of the terrain engine is based on code from the Demeter Terrain Visualization Library by Clay Fowler, 2002. I have ported from its original OpenGl implementation to DirectX, and improved it in many ways. The terrain engine uses ROAM or Real-time Optimally Adapting Meshes algorithm.

A global terrain can either be single or latticed. The latter will load terrain as tiles automatically when they are potentially visible. A global terrain's world coordinate (x,y) is defined as below: x>=0, y>=0. A single terrain usually has less than 512*512 vertices and is suitable for terrain size smaller than 2000*2000meters. A latticed terrain has no theoretical upper limit of terrain size. In a latticed terrain, each terrain tile may be 500*500meters (possibly containing 128*128 elevation vertices) and there may be any number of such tiles in the latticed terrain. For example, if 64*64 tiles are available on hard disk, then the total terrain size will be 32000*32000 meters. Internally, terrain tile is dynamically loaded into a sorted map. Therefore, there is an upper limit as for the total number of tiles loaded concurrently. But in most cases, only the 9 tiles around the camera position need to be loaded for rendering. One can also specify how many tiles to be cached in memory. By default it is set to 18.

For each single terrain tile, the main or base texture is automatically broken in to texture tiles (each of size 256*256). A common texture is repeatedly (tiling) rendered on another layer on top of base texture. However, the main or base texture does not have enough resolution and, in most cases, is used for rendering terrain in the far side. A matrix of high-resolution textures is used to rendering terrain surface near the camera position. For a 500*500 meter terrain, there is usually 16*16 numbers of high-resolution images covering the entire terrain. If the high-resolution texture is absent, the base texture and the common texture are blended together to give a fair look of the terrain surface. In the future, I will support alpha mapping (light maps) to blend several textures to the same terrain surface. Alpha mapping will significantly reduce the amount of high-resolution texture files used.

The main game engine communicates with the terrain engine through the CGlobalTerrain Class. Figure 4.7 shows the collaboration class diagram. Global Terrain is loaded from text based configuration files. There are two kinds of configuration files. See below.

- Configuration file for a single terrain.
- Configuration file for a lattice based terrain.

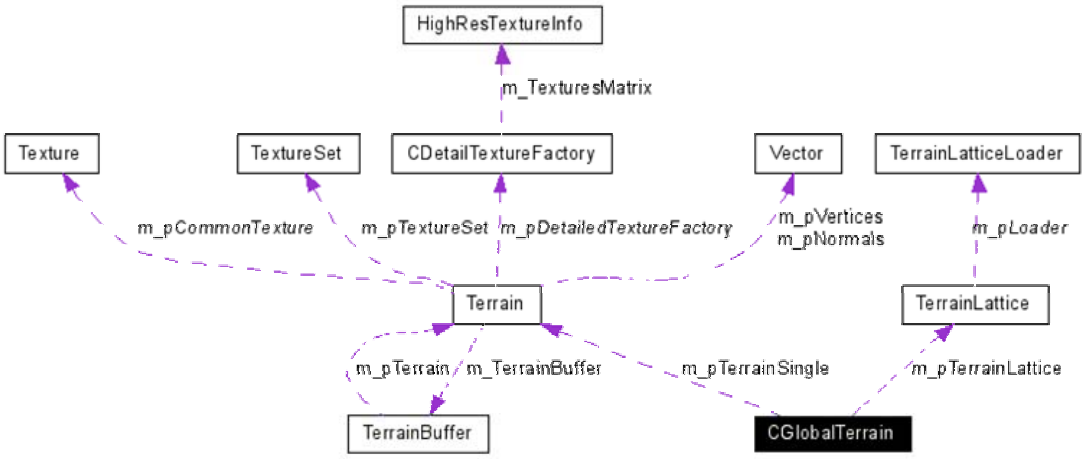

Figure 4.7 Collaboration diagram for CGlobalTerrain

4.3.3.2 File Format: Single Terrain Configuration

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:

```
-- 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.jpg

-- 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 = 533.3333

-- 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

-- 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 = 16
-- Number of texture files used.
NumOfDetailTextures = 3
texture/data/detail1.jpg
texture/data/detail2.jpg
texture/data/detail3.jpg
-- Format: (matrix.x, matrix.y)={nLayer0}[{nLayer1}][{nLayer2}][{nLayer3}]
-- nLayer{i} is the index into the detail texture list for the i th layer
(0,0) = {1}{2}
(2,4) = {1}
(3,5) = {2}
(3,15) = {0}{1}{2}
```

4.3.3.3 File Format: Latticed Terrain Configuration

This is the 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

```
type = lattice
TileSize = 533.333
-- 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
```

4.3.3.4 File Format: Para-Raw Terrain Elevation file

The file name must end with ".raw", for example. in the single terrain configuration file: Heightmapfile = terrain/data/LlanoElev.raw

The engine will assume that the elevation file is of RAW terrain elevation, otherwise it is treated as a gray scale image. In a gray scale image RGB(0.5,0.5,0.5) is of height 0. However, gray scale image has only 8-bits levels, which is insufficient for most application. That is why I implemented the RAW terrain Elevation file.

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);

4.3.4 The Ocean Manager

The Ocean Manager in ParaEngine implements a real time level-of-detail deep-water animation scheme, which uses many different proven water models. In short the animation model is based on mixing the state-of-the- art water models from the computer graphics literature to suite our need for it to remain real time. This includes:

- Ocean graphic statistics based surface wave model for ocean waves (FFT)
- Physical correct surface wave model, taking depth into account, for realistic shorelines etc. (Shallow water waves)
- View dependent water coloring (Color of water)
- Global reflection/refraction (Reflection/Refraction)

The mathematics of the model can be found in the famous water rendering tutorial [25] and the paper [26]. The wire frame implementation is shown in Figure 4.8

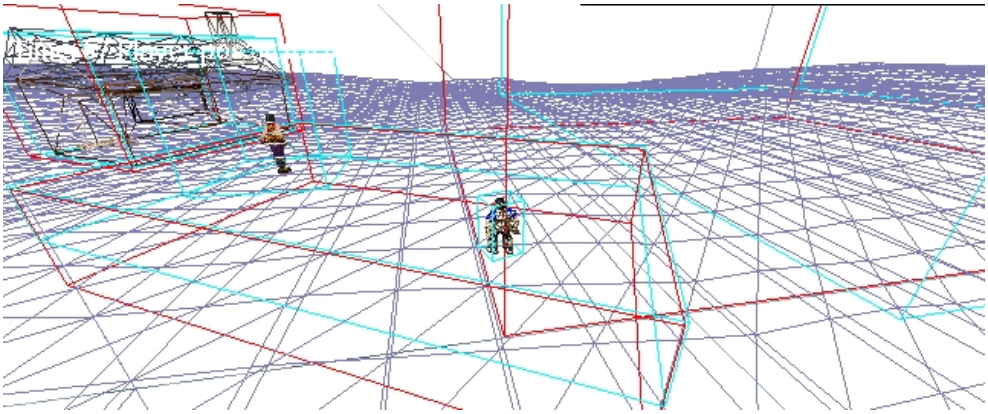

Figure 4.8 Ocean Manager in ParaEngine

4.3.5 Mesh and Animation File Support

This section covers all the design issues about the mesh and animation file support in the ParaEngine. ParaEngine defined its own file format called ParaX format. It is built on top of Microsoft X file template. It is also designed to be compatible with DirectX V9.0c D3DX API. ParaX format defines hierarchical frames (bones), skinned meshes, bone animations, animation sequences, speed of animation, looping properties, bone alpha animations, etc.

4.3.5.1 Related Works

There are many proprietary and free 3D model and animation file formats available. Among the most popular are 3DS, MD2, MD5 and X. 3DS is kind of old and does not explicitly support skinned animation. X file format is primarily used through the DirectX retained mode API. It directly supports skinned animation and has both text and binary formats. DirectX has .x-file interfaces that provide a simple and safe way to import data into DirectX applications. These interfaces identify structures with unique identifiers (IDs), validate data

layout, and provide data referencing and other intrinsic features. Moreover, X file is extensible for arbitrary nodes to be attached to its tree based file structure. Yet, X file is not commonly used in actual game development, perhaps, because that it has not been fully open-sourced to this date. [27] gives the X file specification. [28] presents a more concrete example of loading X file based skinned animation with DirectX retained mode API. There are several exporters including those open source ones provided by DirectX for Discreet's 3ds max and Alias' Maya.

4.3.5.2 ParaX File Specification

The new file format is built on top of Microsoft X file. It is also designed to be compatible with DirectX V9.0c D3DX API. ParaX format defines hierarchical frames (bones), skinned meshes, bone animations, animation sequences, speed of animation, looping properties, bone alpha animations, etc.

The DirectX File Format provides a rich template-driven file format that enables the storage of meshes, textures, animations and user-definable objects. Support for animation sets allows predefined paths to be stored for playback in real time. Also supported are instancing which allows multiple references to an object, such as a mesh, while storing its data only once per file, and hierarchies to express relationships between data records. The DirectX file format is used natively by the Direct3D Retained Mode API, providing support for reading predefined objects into an application or writing mesh information constructed by the application in real time. For more information about the X file, please see it specification [27].

4.4 ParaX Templates

The ParaX templates consist of the original DirectX retained mode X-file templates and the following templates (Table 4.1).

Table 4.1 ParaX Template

```
xof 0303txt 0032
template Anim{
 <00000002-0000-0000-0000-123456789000>
 STRING name;
 DWORD from;
 DWORD to;
 DWORD nextAnim;
 FLOAT moveSpeed;
 Vector minEntent;
 Vector maxEntent;
 FLOAT boundsRadius;
}
template AlphaKey {
```

```
 <00000012-0000-0000-0000-123456789000>
 DWORD nKeys;
 array TimedFloatKeys keys[nKeys];
}
template Alpha{
 <00000011-0000-0000-0000-123456789000>
 [...]
}
template AlphaSet{
 <00000010-0000-0000-0000-123456789000>
 [...]
}
template Sequences{
 <00000001-0000-0000-0000-123456789000>
 [Anim <00000002-0000-0000-0000-123456789000>]
}
template ParaEngine{
 <00000000-0000-0000-0000-123456789000>
 [...]
}
template XSkinMeshHeader {
 <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
 WORD nMaxSkinWeightsPerVertex;
 WORD nMaxSkinWeightsPerFace;
 WORD nBones;
}
template VertexDuplicationIndices {
 <b8d65549-d7c9-4995-89cf-53a9a8b031e3>
 DWORD nIndices;
 DWORD nOriginalVertices;
 array DWORD indices[nIndices];
}
```

```
template SkinWeights {
 <6f0d123b-bad2-4167-a0d0-80224f25fabb>
 STRING transformNodeName;
 DWORD nWeights;
 array DWORD vertexIndices[nWeights];
 array FLOAT weights[nWeights];
 Matrix4x4 matrixOffset;
}
```

The retained mode templates are used for defining static meshes, textures, norms, frames (bones), skins and animation key frames. They are fully compatible with the X file format. The additional template is used for defining game engine specific data, such as object speed, animation looping index, bounding sphere, animation name, etc.

The top-level objects in ParaX file are demonstrated in Figure 4.9. In the figure, three top-level objects are defined. The first two are compatible with DirectX retained mode API. The last one is defined by ParaEngine. Since there is no overlapping between these top-level objects, ParaX file can be viewed or edited by any DirectX supported commercial software, such as Right Hemisphere's Deep-exploration [29].

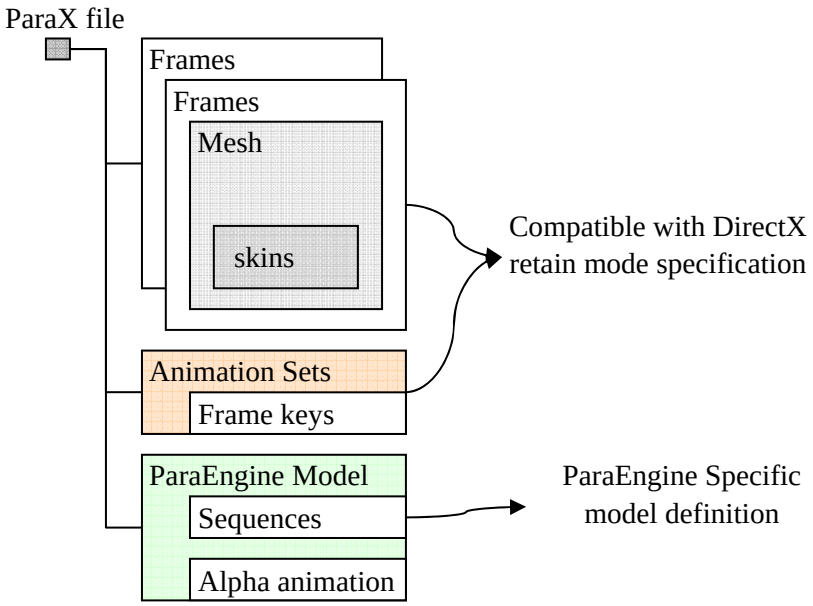

Figure 4.9 ParaX file Hierarchy

4.4.1.1 ParaX File Implementation

Please see the reference manual (ParaEngine, 1.2) for class information. From an external viewpoint, the ParaX is implemented as shown in Figure 4.10.

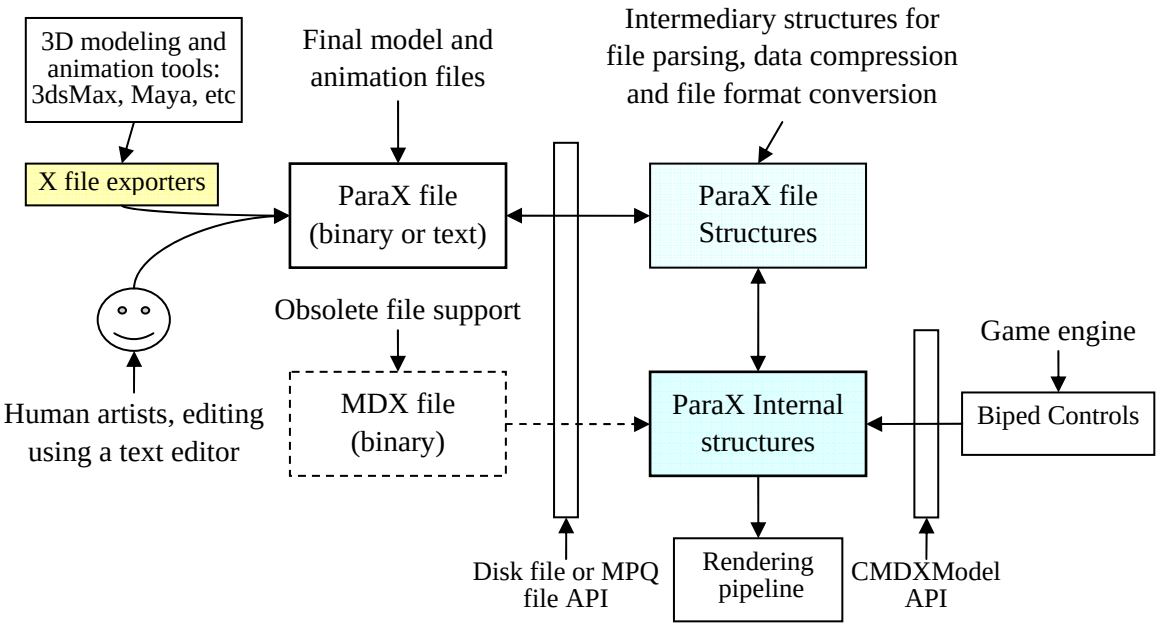

Figure 4.10 ParaX file implementation pipeline

It begins by modeling 3D models in any commercial modeling tools, such as 3dsmax or Maya. It is also possible to make skinned animation sequences in these modeling environments. Then, use any X file exporter to save the model and animation data in x file. There are many X file exporters including those provided by DirectX SDK. I suggest the one provided by Right Hemisphere [29]. The artist must edit the x file (in text format) according to the ParaX file template defined in Section 2, such as adding speed, named sequences, bounding sphere, etc. After that step, the x file is ready to be digested by the game engine. The X file can either be saved in MPQ file (our compressed-file cluster manager) or in the disk file directory. The game engine will then parse the file in to intermediary structures, then in to the ParaX internal structures. Intermediary structures are released immediately when the internal structure is created. Internal structures are used by the game engine to render the mesh in a certain bone pose. The game engine or the biped controller can modify the Internal structure through CMDXModel API. These modifications usually include changing character speed, model color, scale, animation, poses, position, etc.

4.4.1.2 Exporting and Using ParaX Files

This section continues with Section 3. Please read the above section before going on. The mayor tasks of the artist are modeling the characters with an animation tool, exporting them as an X file in text format, then, adding game specific information to the file with a text editor.

It is worth mentioning that multiple animation sets in ParaX file are automatically concatenated. Each animation should be no longer than 10 seconds. However, it is safer to use only one animation set in each ParaX file, and no restrictions on animation length is imposed in this case. Currently all X file animation exporters can only port one animation set at a time. So to ease the task, artist can export all animations in a single file. For example, in 3dsmax, artist can build a character model, bones and a long sequence of animation with all walking,

running, attacking sequences. The artist then writes down the starting and ending frame numbers for each of these sequences. In its x file, artists can add text similar to those in Table 4.2 to tell the game engine such information.

Table 4.2 ParaX File Sample

```
// …
// other exported text above, such as bones, mesh, animations, etc
// …
ParaEngine female1{
 Sequences{
  Anim{
   "Stand"; // sequence name
   120; // start frame number
   2880; // // end frame number
   -1;     // loop on its self
   0.0;    // no speed
   0;0;0;; // bounding box min
   0;0;0;; //// bounding box max
   100.0;                    //bounding sphere radius
  }
  Anim{
   "Walk";
   100120;
   102880;
   -1;     // loop on its self
   25.0;   // speed
   0;0;0;;
   0;0;0;;
   100.0;                    //bounding sphere radius
  }
  Anim{
   "Attack1";
   400000;
   403840;
```

```
  -1;     // loop on its self
  0;      // no speed
  0;0;0;;
  0;0;0;;
  100.0;                    //bounding sphere radius
 }
}
AlphaSet{
 }
}
```

Special attention needs to be paid to the character unit scale and orientation. No matter what system unit artists use when modeling the character, it must be eventually converted to centimeter and left-hand coordinate system, with the character facing the positive y axis and height in positive z axis as shown in Figure 4.11. Animation key frames should be measured in milliseconds (i.e. 1000 means 1 second). Figure 4.11 is a 3D animated character model based on a demo ParaX file I created. One can find it in the game engine's directory: *xmodels\moon_female.x*. Please pay attention to how the model is posed in the world coordinates. The model's feet are on V (0, 0, 0).

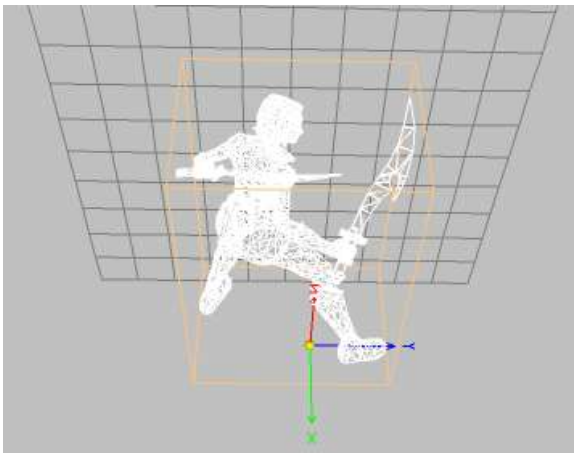

Figure 4.11 3D Model orientation in ParaX file

The artist can also test their models in the game engine to preview its effect. ParaEngine script provides an easy mean to load ParaX file and create character scene objects based on it. Actually it is as easy as a few lines of code. See below.

```
ParaCreateAsset("MA", "Hero", "xmodels\\moon_female.x"); -- load the ParaX file model: Hero
ParaExecuteCmd("init");
ParaCreateSceneObject(nil, 63, "friend1", "Hero", 45,50,0, 0.35); -- create a biped scene object
ParaAddEventToObject("friend1", "spds", 2.0);  -- set speed scale of current biped
```

The above code can be found in \script\scenes\loadscene1.lua of the ParaEngine directory. Actually artist only need to replace the text in bold with the current file name or override the file with the current one. The above script will be loaded when the game is started and test game button is clicked.

Shadow is supported in ParaX file based mesh. Please refer to the special effect document [self: 8] for more information about shadow support in the game engine. Generally, artist should use closed mesh in order for the model to cast correct shadows. Objects with transparent textures (such as trees) need to enforce Z-fail algorithm. This is done by adding an additional parameter to its mesh creation API. See below.

```
-- create asset enforcing shadow capping
ParaCreateAsset("MA", "tree0", "Doodads\\Terrain\\AshenTree\\AshenTree3.mdx", nil, 1);
```

4.5 Mobile Scene object Presentation

4.5.1 Biped Object

CBipedObject class can be used to represent biped object (like human, re spawning monsters) in the scene without inheriting and adding new functions.

This class and its inherited classes are capable of some basic tasks to make the object (character) aware of its surroundings, as well as carry out all actions and movements.

CBipedObject provides many virtual functions that may be called many times by other modules(such as Environment Simulator and AI Simulator), so that inherited class that override the parent implementation easily.

The biped can be associated with either MultiAnimationEntity or MDXEntity which will used to render the 3D animations. One of the AnimationInstanceBase derived class should be used to control the animation of the biped object as well as the speed of the biped. AI controllers can also be associated with biped object, in which case the biped would act on their own automatically. The behavior of the biped is controlled through Events like all base objects. High Level Event can be assigned to biped object to control almost everything about a biped object at run time, such as playing new animation, walking to a new position or even assigning AI tasks. Figure 4.12 shows the class diagram.

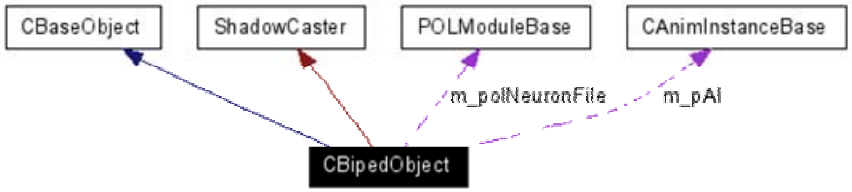

Figure 4.12 Collaboration diagram for CBipedObject

4.5.1.1 Scene object event

A scripting event is a string command to create, modified or controls a unit in the scene. It can either be issued by the program in the engine or read from a plot script.

CBaseObject can receive events from its caller; those events will be added to each object's event pool and be processed by the environment simulator and neuron network simulator.

Scripting events are not processed by a single daemon in a single place. Instead, each scene object must parse those string events and execute them in their own right.

Currently there are two layers of events:

- High level event (or intension event): HLE
- Low level event (or resultant event or God event): LLE

HLE are events that tell a caller's intension of the object. For example, "go forward", "fight XXX", "turn left", "speak XXX". LLE are events that that are generated from HLE or given by a "God" of the world. For example, "be at (0,1,0)", "face (1,0,0)", "idle", "walk";

Table 4.3 Event layers

	Examples	**Generator**	**Method of removal**
HLE	go forward	User Control, AI engine, Script sequence	Release upon finish
LLE	face (1,0,0)	Environment simulator, Network commands, Script sequence	Command conflict detection

Event processing pipeline:

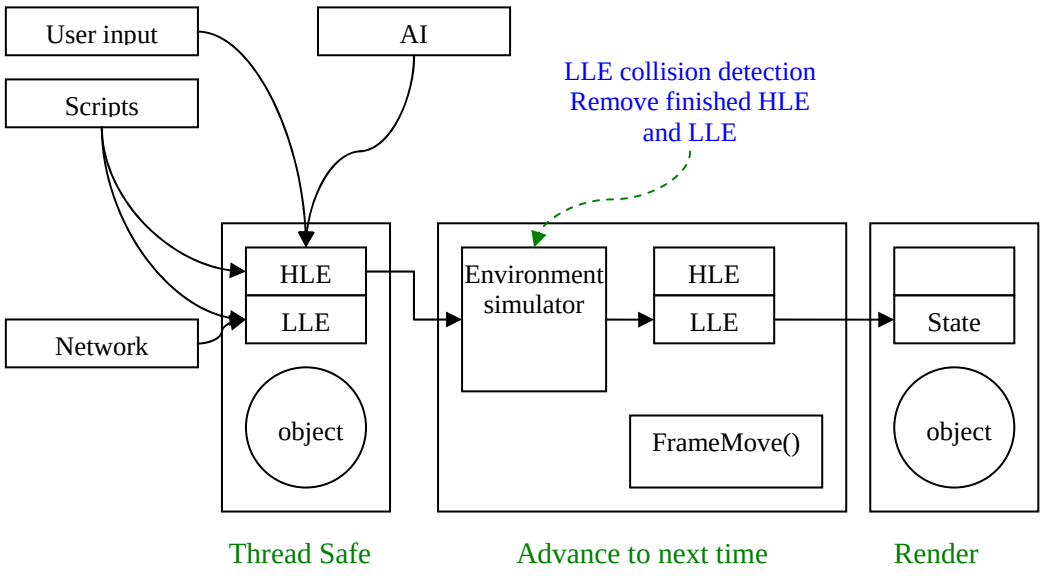

Figure 4.13 Event processing pipeline

4.5.2 Other Global Objects

There are also other miscellaneous objects, such as sprites and 3D text.

4.6 GUI and Effect Objects

4.6.1 GUI Objects

Like 3D scene objects, GUI objects are also arranged in a tree hierarchy with the top level object called CGUIRoot. CGUIRoot expose a rich interface to the scripting system to build the GUI tree. For example, one can insert sounds, text, buttons, windows, textures, mouse or key sensors, etc to the GUI tree. It is similar to generic Windows Programming. Yet it uses a much casual and simple programming interface. Figure 4.14 show the GUI root object.

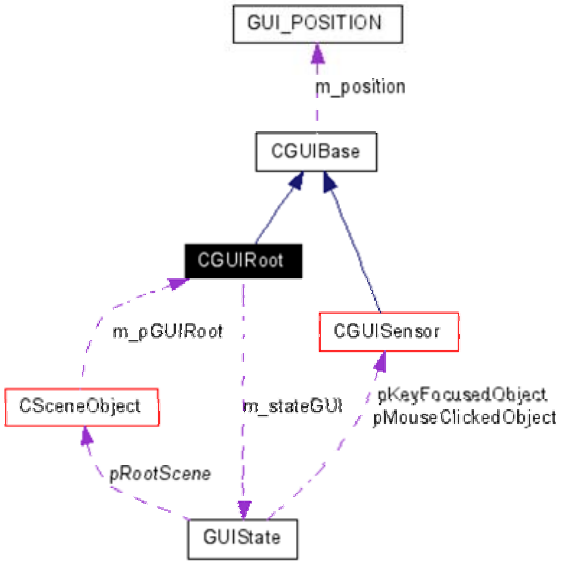

Figure 4.14 The GUI Root Object

4.6.1.1 Overview

The main difference between the GUI object and 3D scene object is that (1) 2D GUI object are not tested against view frustum, instead it is controlled by visibility tag automatically or through user input. (2) 2D GUI object generally does not obey the physical law of 3D world. (3) GUI object are generally specified by screen coordinates, instead of 3D position. (4) GUI object may be frequently created and discarded.

There are two types of GUI objects defined in ParaEngine (1) status objects (2) GUI sensor object.

- Status object are solely used to render annotations to the 3D world object.
- GUI sensor object will trigger Script file, once the user clicked or typed in its client area.

Here are some examples of using GUI objects.

- A health bar at the bottom may be attached to the 2D screen; a dialog box (including the texture of a button) is attached to the center of the 2D screen. A line of text is attached to the 2D screen too, or possibly attached to another status object (a dialog box for instance). Background music is attached to the screen.
- A name text is attached to a biped object in the virtual world. A 2D sprite animation is attached to a biped object in the virtual world. A 3D mesh and a texture animation

(rendering a magic) is attached to a biped object in the virtual world. A sound track is attached to a biped in the virtual world.

- A particle system is attached to the screen/ or a single building/biped in the virtual world.

Figure 4.15 shows GUI objects that are currently supported in ParaEngine.

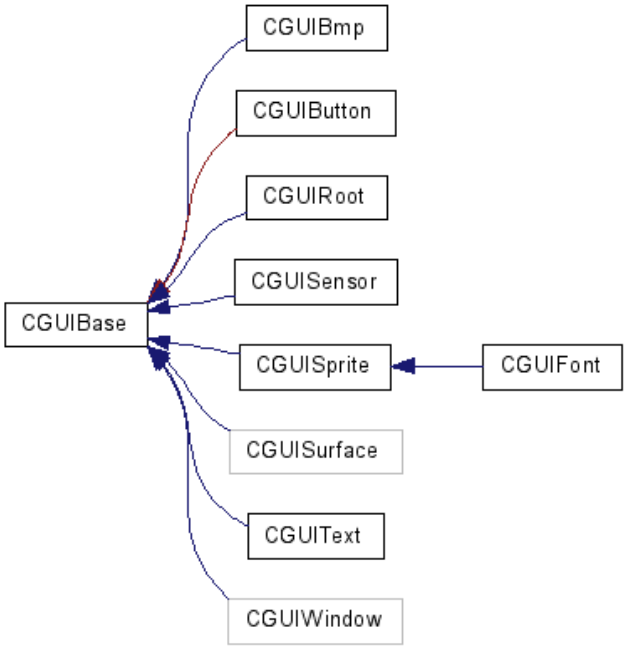

Figure 4.15 GUI objects

4.6.1.2 GUI Pipeline

When rendering scene, root scene and the root GUI are rendered in sequence. What GUI objects do is making screen annotations in the 3D scene. For example, some character's head-up display is created with GUI objects. Text, sounds and even dialog boxes can be attached to the scene objects or the screen. Sensor based GUI objects will also accept user input such as key strokes and mouse clicks. Sensor object is always associated with a script file, once a sensor is triggered, its corresponding script file will also be activated. Figure 4.16 shows a working GUI snapshot.

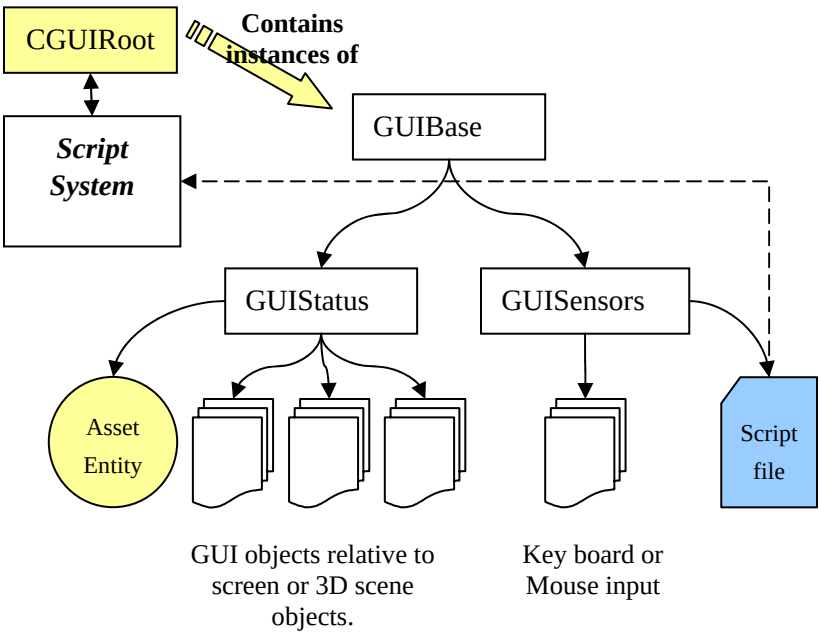

Figure 4.16 GUI Objects Snapshots

4.6.2 Projected Textures

It is not yet implemented.

4.6.3 Particle Systems

It is not yet implemented.

4.6.4 Sounds

It is not yet implemented. Currently, ParaEngine supports a global sound manager for playing background music or sound clips.

4.7 Graphic Rendering Pipeline

4.7.1 Rendering Pipeline Overview

Much of work in scene rendering has to do with transversing a scene graph and selectively draw its nodes in a certain order. An open source game engine [30] is named after it. The details of a rendering pipeline can differ greatly for different game engines. I will present here one pipeline used in my game engine. It continues with the Render Scene function in Table 3.1. The pipeline presented is suitable for both outdoor and indoor rendering. Please see Table 4.4.

Table 4.4 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 landscape{
        Build LOD terrain based on current viewport and camera settings.
        Draw the terrain, water, tree, grass: we may delay their rendering for occlusion
        sorting.
    }
    For each global objects in the scene graph{
        Perform rough clipping test using its bounding shape, such as box and sphere.
        Add to post rendering list or animated biped list or sprite object list according to its
        type.
    }
    Transverse quad tree scene graph{
        For each scene object{
            Perform rough clipping test using its bounding shape, such as box and sphere.
            Switch (object. type){
                Octree, BSP or Portal nodes (usually level mesh):{
                    do occlusion test
                    draw the mesh
                    Add any scene objects in its leaf nodes to post rendering list
                }
                Other small sized mesh nodes: { add to post rendering list }
                Animated biped nodes: { add to animated biped list }
```

```
                    Billboard or sprite objects: { add to sprite list }
                }
            }
        }
        Transverse post rendering list {
            Sort by texture, alpha or rendering state
            Perform rough occlusion test with depth buffer using its bounding shape.
            Draw the object: it may perform future optimizations internally.
        }
        Sort and draw object in sprite list
        Advance animation in animated biped list (RENDER_TIMER)
        Draw shadow casters in animated biped list: only cast shadows on object rendered previously
        For each object in animated biped list{
            Perform rough occlusion test with depth buffer using its bounding shape.
            Draw the object
        }
        Render dummy objects for debugging purposes
}
```

A good 3D rendering pipeline can be divided into clipping, culling, occluding and computing the right level of detail (LOD). Different game genres may emphasize certain components to reach the desired performance. Generally, it falls into three categories [31]: outdoors, indoors, and planetary (outer space). The first two are commonly used. 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 [32] 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.

As one may notice from the rendering pipeline in Table 4.4, it combines outdoors and indoors rendering, by ordering them in a certain way. Generally speaking, I render outdoor scene followed by indoor scene, postponing the rendering of any small-sized but high-poly object until large objects which may occlude them have been drawn. Shadow casters are rendered only before objects that do not receive shadows.

4.7.2 Indoor and Outdoor Rendering

Popular indoor rendering techniques include:

- 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): It is a 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 paint only what is effectively seen. Unlike BSP, the world hierarchy is not automatically computed, and visibility is computed in real time, not preprocessed. It is also the most difficult one of the three methods discussed.

Outdoor rendering techniques mainly deal with large areas of landscape or terrain rendering. An extensive overview of terrain rendering techniques is given in [33]. Popular outdoor techniques include ROAM, Chunked LOD, etc [32] [34]. Almost all outdoor rendering techniques are relatively difficult to implement.

4.7.3 View Culling Techniques

Please see Chapter 4.3.2 Fog and Object Level Culling for the relationship between fog and object level culling. In ParaEngine, object level culling is mainly performed on a quad tree. Object view culling which is carried out by the root scene object will produce a list of objects whose bounding boxes intersect with the view frustum. The rendering pipeline then sort these objects according to their rendering methods or their distance to the camera position (front-to-back). Finally, the rendering routine of each object is called. The render function of each scene object may use a different hidden surface removal technique, such as BSP, PVS, etc. For most small or medium sized scene objects in ParaEngine, a brute force rendering is used. For terrain and ocean tiles, some LOD is used to further reduce the number of triangles sent to GPU.

4.8 Shadow Casting

ParaEngine is initially designed for outdoor games. Light sources (most likely directional light like the Sun) will cast shadows from a large number of static and dynamic geometries in the scene, such as trees and characters. Some special effects such as magic and character selection circle will also involve casting shadows on the ground from invisible cookie objects. This section covers shadow rendering issues in ParaEngine.

Shadows are important to image synthesis because they add realism and help the viewer identify spatial relationships. 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 such as in interactive computer games. Existing methods are usually limited by sampling artifacts or scalability issues. The choice or design of shadow rendering technique in the game engine is crucial for its performance.

4.8.1 Previous Works

Since a vast number of hard and soft shadow methods [35] [36] [37] exist for general and very specific situations, we will only briefly discuss some methods here, focusing on those suitable for interactive and real-time applications. As a good starting point, please refer to Woo et al.'s paper [38] for a broader survey of shadow algorithms. Table 4.5 shows a comparison of some commonly used real-time shadow rendering algorithm.

Table 4.5 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

4.8.2 Shadow volume

Morgan [39] proposes an optimized shadow volume algorithm tutorial. The implementation of shadow volume [40] in the game engine is largely based on this tutorial. I will not repeat the theory behind this algorithm in this article. A recent version of the algorithm can also be found at [41]. The major reason I choose shadow volume is that it is a general algorithm suitable for casting shadows from closed geometries. It is also possible to use some image filters (convolution) to blur the shadowed region in stencil buffer, in order to render it as fake soft-edged shadow.

4.8.3 Limitations to Shadow Volume

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 you 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 are geometric complexity (and number of meshes) and light count, the best systems will use an algorithm to select only the most important lights, and only the affected geometry. In our game engine, I generally enable only one light for each shadow caster.

5. Problems When the Camera Intersects the Shadow Volume

There are quite serious issues with camera movement when using Depth-Pass shadow volume rendering. When the camera is positioned inside a shadow volume you will get noticeable artifacts appearing on-screen. There aren't any good solutions to this problem apart from switching to another, similar, rendering algorithm: Depth-Fail. A combination of depth fail rendering, shadow volume capping and projection tweaks allows for a robust shadow rendering procedure.

4.8.4 Implementations

There are two kinds of scene entities that may cast shadows. One is static mesh entity; the other is ParaX file based skinned animation entity.

Shadow volume algorithm imposes an additional constraint for models that are animated by skinning (the process of blending different transformations for 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 implemented in software, so if shadow volume is enabled, it becomes the only rendering 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. sky box
2. terrain object
3. sprite object
4. static mesh object
5. **shadows** of ParaX file based biped object
6. ParaX file based biped object
7. GUI objects

Since shadows will only be cast on objects rendered previously in the pipeline, ParaX file based biped objects will not cast shadows on themselves (we draw shadow casters after their shadows). Due to the nature of the shadow volume algorithm, the transparent region in transparent textures will appear to be shaded on the 3D model. A close-cap shadow volume will fix the problem; however, rendering (front) caps as in the Z-fail algorithm is computationally expensive. We use z-pass wherever possible.

Shadow Volumes Calculation

Any shadow caster class implements the ShadowCaster interface. In ParaEngine, the biped scene object is one such shadow caster. In its shadow caster implementation, it calls the Build Shadow Volume function of its associated ParaX model class. It is the model object which has the information of the geometry of the 3D model that will 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 4.17.

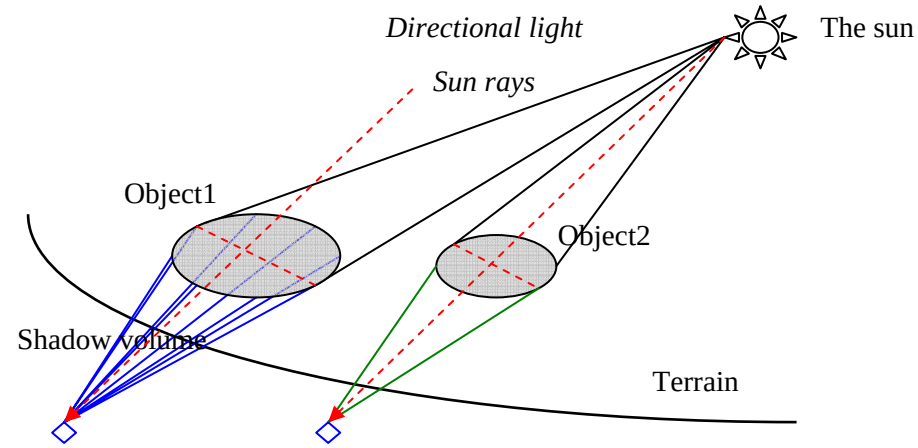

Figure 4.17 Shadow Volume Calculation for Directional Light

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 4.6 is used decide when to swap to z-fail algorithm. The rules are labeled from (a) to (e).

Table 4.6 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 4.18 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).

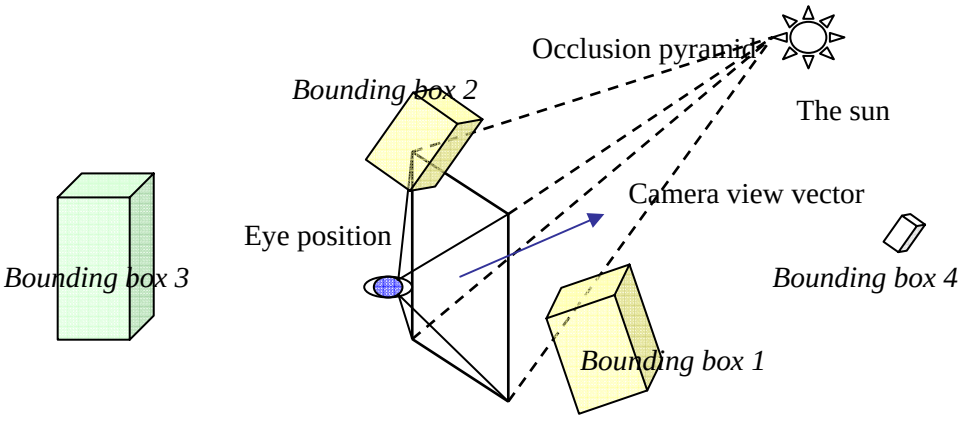

Figure 4.18 Z-Fail algorithm testing

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 huge consumer of invisible fill rate. One obvious way to alleviate the CPU crunch 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. Also another good trick is to pre-calculate all static shadow volumes. that is, shadows generated by lights and objects that do not move. Shadow extrusion may also be done in shader language HLSL.

4.8.5 Sun Light Emulation

Sun light is a directional light which will cast shadows for scene objects. In additional to its visual effects, it also provides cues to users about the orientation. Sun light is implemented as part of the sky entity. So far, sun light is the only supported light (directional) that will cast shadows.

The direction of sun light can be changed indirectly by modifying the time of the day. A typical case is to advance the time of day in the sun light object in each frame move call.

4.8.6 Implementation results

Figure 4.19 shows the rendering results.

The testing platform is Mobile Radeon 7500 graphic cards with 2.4G CPU. Our game can maintain 30 FPS under 640*480 screen resolution for 90 percent game scenes. The current shadow casting architecture is robust enough for the current game configuration and that it is fairly fast to main interactivity.

Figure 4.19 Shadow rendering results

4.9 Conclusion

Most techniques in this Chapter are common in other game engines. However, their combination makes this rendering engine unique. The proposed rendering engine is suitable for large scale outdoor and indoor mixed scenes, as well as extensible game scenes.

Chapter 5
Simulating the Game World

5.1 Physics Engine

5.1.1 Introduction

This chapter covers the design issues about the integration of a third party physics engine into the existing ParaEngine physics system. The chosen physics engine is Novodex Engine [9] under public license. However, ParaEngine does not rely completely on Novodex for its physics; instead it adopts a dual physics system, where it automatically swaps between these two physics systems when certain conditions are met. Generally speaking, the original physics engine handles all physics when character is situated on the global terrain; whereas Novodex handles physics when the character is inside or in touch with a more complex artificial environment, such as a bridge, castle or cave. For places where the terrain meets the artificial environment, some rules are used to swap between these two physics system or just block the character. This article will cover the updated environment simulation pipeline in the game engine, such as when and how physics data are loaded. It will also succinctly introduce the method of building complex physical environment from ordinary shapes and arbitrary triangle meshes.

5.1.2 Review of Physics Simulation in Game Engine

Although a game engine can directly implement the entire simulation framework, it is common these days to use a middleware physics engine [9] [10] for handling the portion of physics which requires more complex simulation (such as dynamic rigid body simulations). [11] provides a comprehensive overview about a wide range of problems when outsourcing the physics engine in game development.

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.

In game development, we usually have many hand-animated game characters, complicated machines and some moving platforms. These objects are not obligated to obey the laws of physics. So the third 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 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. And game developers need to manually tell the engine how the objects should respond.

5.1.3 Novodex Physics Engine

In [42], a full integration guide of the Physics engine is given. All physics SDKs are built upon the 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 the rigid bodies are built from, triangle meshes that certain collision shapes are constructed from, 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 the user can employ to query the scene for information. This includes debug wire frames, contact information between designated contact pairs, joint breakage, and the results of ray cast through the scene.

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.

Most actors will be dynamic. They are part of the physical scene and will move and collide with the world and other objects realistically. In general, we want to keep our dynamic object densities within a dynamic range of 10, that is, if our lowest density object has a density of 5, our highest density object should have a density of around 50.

We may make actors kinematic, if we want total control over them. These actors will not react to the world and other actors. They will move precisely where we want them to go, pushing all dynamic objects aside as if they had infinite mass. Kinematic actors should be used for objects we consider to be of such high density, they will push all other objects aside. They are good for moving objects that are effectively immune to the physical scene, like heavy moving platforms or large moving blast doors or gates. Kinematic objects can be turned into dynamic objects and vice-versa. If the objects are stationary and will never move, we want to make them static. Static objects cannot be made dynamic or kinematic.

To summarize, the physical world is built with dynamic actors 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 all other dynamic objects asides. 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 in the Novodex engine.

5.1.4 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 dynamic 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 advances 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 Novodex may report user contacts through callbacks (since, Novodex is multi-threaded; these callbacks are in different thread than the game loop). The game engine should

remember these contacts in the callbacks; so that it can see them and respond to them when the Novodex thread is finished and the game loop thread receive the control.

Now we will see separately how physics in handled in both threads. The following code gives an overview of what should be done in the game loop.

```
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).
}
```

For the motion of kinematic object, it is possible to understand the layout of the environment just using contact sensors and tracking the position; terrain learning is required for collision prevention. However, this task is less realistic because it relies on trial and error rather than prediction, and a character finds out about obstacles only when it is too late. The shape of the sensor used in collision detection can be point, line or event the bounding shape of the character itself.

In the following sub sections, we will look at three methods for handling kinematics character collision detection and responses. They are shape-sensor based collision detection, single ray casting collision detection, and multiple ray-casting collision detection. These methods are complementary. I have designed them to be used simultaneously in ParaEngine; currently, only the last one is fully implemented.

5.1.5 Single Ray Casting Collision Detection

Ray casting is an old yet still very useful collision detection method. In the simple ray casting physics system, a character's motion is solely decided by whether the ray cast along its path collides with any other world object (Figure 5.1).

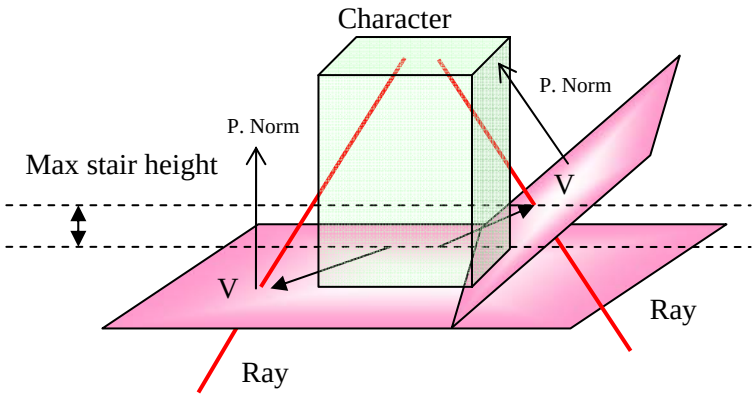

Figure 5.1 Ray casting collision detection

Ray casting should be comparatively fast than using a simplified shape for collision detection, but it is not very accurate in special situations. When using ray casting, all characters in the game engine can use kinematic type actors for their physics. The downside of solely using kinematic actor is that, the motion of kinematic actors can only be controlled by the programmer other than the Novodex physics engine. The good part is that it is faster to calculate and its motion is predictable and quite controllable.

5.1.5.1 Single Ray Casting Sliding Wall Motion

The ray casting vector is modified to reflect the distance that the character was able to move before it collided with the world. So we should move the character that far, and subtract the new vector from the original vector, to get the distance still to go. We will call this V, as shown in Figure 5.1. A special condition is that, the terrain in front of the character is so slope (such as a wall) that it can not walk directly to it. Instead of stopping the character immediately, we may also try sliding the character along the wall, such that it is still approaching the target.

Use Novodex to get the triangle that was collided with. It is described by the colliding point P and the norm of the colliding face at point P, denoted by P. Norm. We want to give our vector a new direction so that it can walk toward its target position. The cross-product (V × P. Norm) of the two will give a vector which is on the colliding plane, but perpendicular to the vector that we want. Taking the cross-product again (- (V × P. Norm) × P. Norm) will give the vector we want. Now we try to move the mesh along this new vector. Hopefully, there will be no collision this time, but if there is we must do the whole thing over again. We should also prevent it from running into an infinite loop if the player walks into a corner.

5.1.5.2 Single Ray Casting Physics Architecture

With ray casting, the new physics architecture can be completely merged into the original architecture. See Table 5.1.

Table 5.1 Environment simulation step function using ray casting

Environment **simulation** (SIM_TIMER) {

Fetch last simulation result of dynamic objects

Validate simulation result and update scene object parameters, accordingly.

Perform rough collision detection using the quad tree based terrain, save the result to active biped list. A snapshot of Active Biped is given below. Objects that rely on static actor for its physical property are regarded as transparent object in this simulation pass.

```
struct ActiveBiped {
        /// the biped scene object.
        CBipedObject* pBiped;
        /// terrain tile that tells most exactly where the object is in the scene
        CTerrainTile*   pTerrain;
        /// object list that collide with this biped. It can be solid objects 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;
}
```

So far, biped characters are not moved, nor tested against the global terrain or static actors.

For each biped character in the active biped list {

/// If the biped is in standing state or has just reached its target, it will receive a 0 speed.

V0 = the desired velocity vector of the character.

If(V0 == 0) { continue; }

V = V0; /// the velocity vector to be test with.

solved = false; count = 0;

While (!solved && (count++ < 3)){

cast a ray according to V to the global terrain, static actors and all objects in the collision pairs separately. The intersecting points are denoted by [38]. Note if P_i does not exist, its height is set to negative infinite.

P = $\max_{P_i}\{p_i.height\}$

If(|P . height – Biped . height | < max stair height){

If (|P . norm – (0,1,0) | < max slope) {

Move the biped towards the P from its current position using linear interpolation in the height (y) axis.

}else{

```
                Move the biped towards the P from its current position, without changing its
                height.
            }
            solved = true;
        }else {
            /// try again by sliding along the obstacle.
            V = Normalize(- (V × P. Norm) × P. Norm)*|V0|;
        }
      }
   }
   If(!solved) {
        Ray casting downward to see if the biped is in air, if so let it fall,
        otherwise stop the biped
   }
   Perform AI strategy, etc
}
```

The above architecture performs physics by solely ray casting. All objects including the global terrain, static actor, solid objects, are treated equally in the pipeline. Sliding wall is the default behavior if direct motion is not possible. It is also possible for one character to stand on top of another one. But if the character below moves, the one on top will fall. This is different from an elevator.

5.1.5.3 Modification of the Base Object Interface

Almost all game objects are derived from based object in ParaEngine. The base object offers the following information through a set of virtual functions:

- Object name: m_sIdentifer
- Object type, such as a solid mesh, a biped, a NPC, a OPC, etc: m_objType
- Object volume type, or how the shape of the character should be interpreted, such as solid, transparent, sensor, container, etc: m_dwObjectVolumnType
- Object shape, i.e. is it a sphere, a box, a rectangular (billboard), a circle, an externally defined shape, etc: m_objShape
- All physical properties of the object according to its shape.
- List of child object: m_children
- List of events this object received: m_objEvents

We will pay attention to the object shape and physical properties. In most cases, it is possible to get all the physical property and perform all collision tests with only the base object reference, not necessarily using the actual object reference. This is true even with the introduction of Novodex physics engine. The biggest modification to the old base object interface is that the physical properties should be specified in full 3D environment, (instead of 2.5D).

The object Shape is extended as below:

```
enum ObjectShape{
        _ObjectShape_Circle = 0,
        _ObjectShape_Sphere,
        _ObjectShape_Rectangular,
        _ObjectShape_Box,
        _ObjectShape_NX_dynamic, /// the shape of the object is associated with a dynamic actor defined in Novodex physics engine
        _ObjectShape_NX_static,  /// the shape of the object is associated with a static actor defined in Novodex physics engine
        _ObjectShape_NX_kinematicBox,  /// the shape of the object is associated with a kinematic box shaped actor defined in Novodex physics engine
};
```

The functions (mostly virtual) to access the physical property of the object changes to following:

```
FLOAT GetBoundSphere()
/// get object position in the world space
virtual void GetPosition(D3DXVECTOR3 *pV)
/// return the normal at the terrain position where the object is situated at its x,z plane.
D3DXVECTOR3 GetNormal();
```

5.1.6 Multiple Ray-casting Collision Detection

In this algorithm, multiple sensor rays are used for gathering knowledge of the character's surroundings. BIPED_SENSOR_RAY_NUM defines the number of sensors ray. The higher this number, the more accurate the collision detection will be. Its 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^{th} ray direction = Object facing + (PI / (rayNum+1)) * (i - ((rayNum-1) / 2)

Normally 3 rays will be both accurate and fast enough. See below.

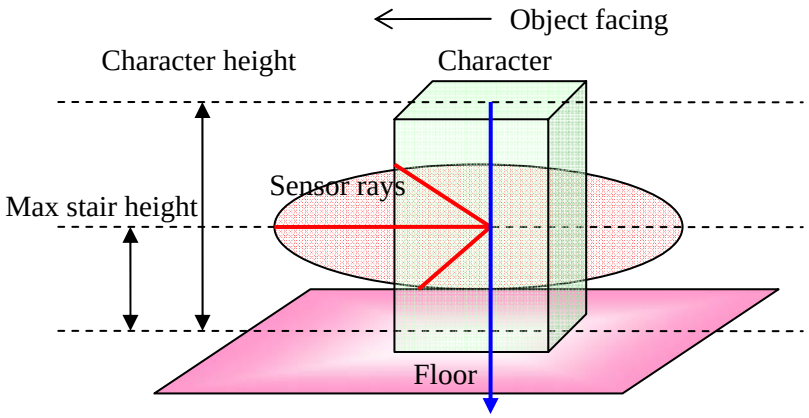

Figure 5.2 Multiple ray-casting collision detection

The length of the sensor ray is the radius of the character; and origin of the sensor rays is the character's global position plus a max stair height, which is usually some factor of the character's height.

5.1.6.1 Multiple Ray Collision Response and Wall Sliding

In single ray casting method, a character relies on the normal of the obstacles to avoid collision. In 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 full algorithm in my implementation is given below:

- If we're within reach, we will stop without reaching the exact target point.
- We cast n (n=3) rays in front of the character, which covers a region of (n-2)/(n-1)*Pi radian. If several sensor rays hit some obstacles within the radius of the character, we will see if the world impact normals are roughly the same. If so, the character is considered to be blocked by a single wall. The wall norm (vWallNorm) is the impact norm returned by the sensor ray whose hit point is closest to the character. In case of a single blocking wall, we will try slide the character along the wall; otherwise the character is stopped.
- We will compute a sliding wall facing as below:
  vFacing = vWallNorm (X) (vFacing (X) vWallNorm); and get a tentative new position(vMovePos) of the character.
- We then cast another ray using the new position to get another impact point. If there is no impact point for the second ray within the radius of (fRadius+m_fSpeed * dTimeDelta), the object is moved to the new position without further processing. Otherwise, from the two impact points, we can calculate the wall direction vector, which will be used for sliding wall. If the current character facing(V), the wall direction(W), and the ray direction (R), satisfies the condition ((R-W)dot(V-W))>=0, the character is still trying to

walk into the wall, we will enforce wall sliding using wall direction vector, otherwise, the character will be allowed to move away using old direction.
- Finally, we will also move the character slightly out of the wall, if it has run too deep in to it
- When the character has been slightly moved in the x, z plane, we will cast another ray from the top of the character downward to get the terrain height below the character. Then we smoothly move the character up the stairs or let it fall with gravity.

5.1.6.2 Implementation result

Figure 5.3 shows characters sliding along small cylinder physics object, climbing up stairs, sliding along lengthy wall mesh, turning shape wall edges, etc. And it is faultless and smooth for all tested situations so far.

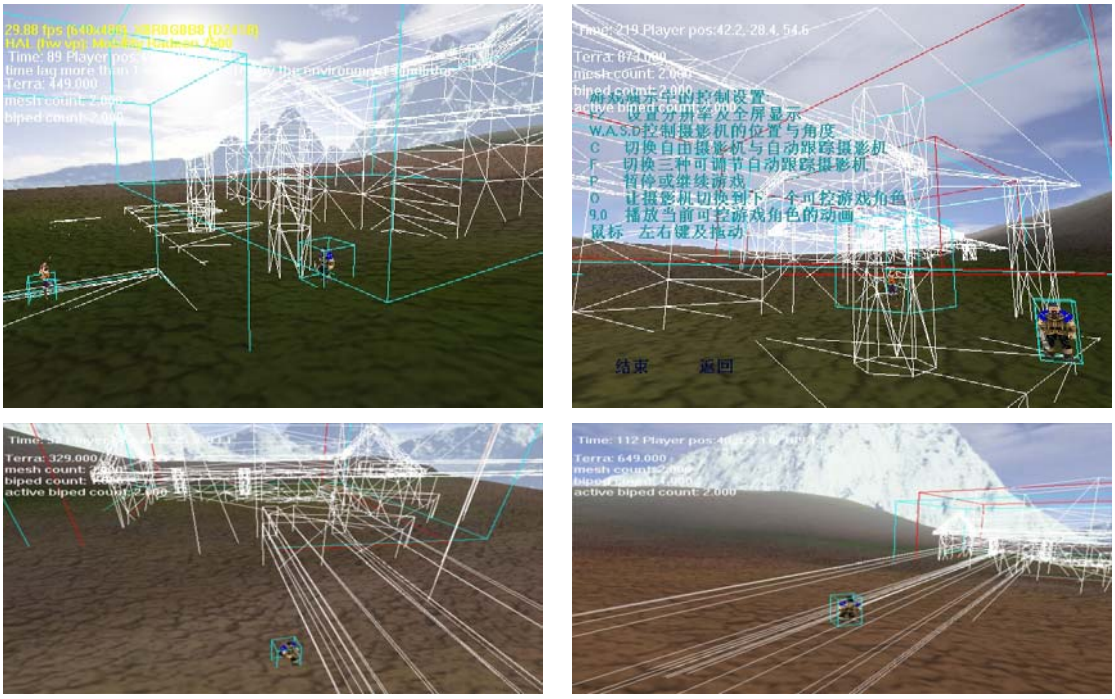

Figure 5.3 Collision detection and response results

5.1.7 Mesh Geometry Creation

Static mesh object is created using the following HAPI functions. Please refer to the NPL reference guide for more information.

```
ParaAsset.LoadStaticMesh("cube", "xmodels\\sample.x");

ParaAsset.Init();

… …

-- create a static physics mesh

player = ParaScene.CreateMeshPhysicsObject ("meshCube", "cube", 6,1,10, true);

player:SetPosition(48, -27.4, 48);
```

```
player:SetFacing(0.5);

ParaScene.Attach(player);
```

The physics mesh creation principles are given in the following sub section. They are provided by the Novodex physics engine

5.1.8 Triangle Mesh Principle

- Be sure that you define face normals as facing in the direction you intend. Collision detection will only work correctly between shapes approaching the mesh from the "outside", i.e. from the direction in which the face normals point.
- Do not duplicate identical vertices! If you have two triangles sharing a vertex, this vertex should only occur once in the vertex list, and both triangles should index it in the index list. If you create two copies of the vertex, the collision detection code won't know that it is actually the same vertex, which leads to a decreased performance and unreliable results.
- Also avoid t-joints and non-manifold edges for the same reason. (A t-joint is a vertex of one triangle that is placed right on top of an edge of another triangle, but this second triangle is not split into two triangles at the vertex, as it should. A non-manifold edge is an edge (a pair of vertices) that is referenced by more than two triangles.)

5.1.9 Physics Scene Principle

The "optimal" setup really depends on the layout of the scene, how you are using the engine, and what platform you are running on (software vs. hardware simulation, if only on PC).

If you split the level up into large cells of triangle meshes, you will reduce the size of the BV tree somewhat, for a light performance gain at the expense of a slightly more expensive broad phase (more actors in the broad phase). If you are running multiple scenes (game play physics in one and effects physics in the other, for example), there may be some additional performance gain due to the fact that these multiple scenes may not block due to simultaneous access to shared resources (the cell BV Trees). Also, with a cell scheme, you are free to load and unload geometry from the scene as needed. If you need to have dynamic terrain, these cells can be made small enough so that you can recalculate the BV Tree at runtime as needed. Composing your static geometry into regular cells reduces opportunities to share mesh representations, which will increase your memory usage and reduce performance, as well.

If you go to the other extreme, however, where you are using very small cells (or an actor for every mesh object, if you are simply matching your graphics instances), you run the risk of generating a massive number of static actors, which will slow down the insertion time of new actors into the scene as the engine finds their place in the broad phase. Once inserted, the performance should be fine, but if you are counting on software simulation (i.e., not using the PhysX chip), these insertion delays will be noticeable as you get into the 20,000-30,000 actor count range (the engine has a limit of 32k, anyways).

So, "optimal" is somewhere between these two extremes. Ideally, you would set your tool chain up so that you can tune the optimal mesh size as your game is reaching completion. At the very least, you should generate a test level similar to your expected game usage (even if it has random content), to determine the sweet spot for your game as a whole.

An example of a potential composition opportunity might be a building that is replicated several times in your level (at least physically--you might have different textures to make

them look distinct). If this building model is composed of a basic shell, and ornamented with hundreds of copies of just a few detail mash models, you would probably want to compose all of these into a single building mesh. You lose the opportunity to share the detail meshes, but you can still share the overall building mesh, and avoid having to add hundreds of individual actors to the scene.

Note: when I say "Compose", I mean add all the triangles of each of the meshes to a single triangle mesh, and make an actor out of that. You would not want to add each of the meshes to the actor as an individual shape, or you will see a performance hit whenever anything interacts with this actor.

5.1.10 Dynamic Object Simulation

Dynamic object is created as global objects and simulated completely by the Novodex engine.

5.1.11 Physics Object Management

There are usually many thousands of physical mesh objects in a scene. It is not possible to simulate them currently. Fortunately, most physical mesh objects are static, so we can choose to simulate object only in the vicinity of mobile physics objects. And a garbage collector routine will remove static physics objects which are far away from the nearest mobile objects. Usually a hit 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.

5.2 Collision Detection

Biped type

There are three kinds of biped objects that act differently during collision detection. I.e. the engine recognized three kinds of bipeds.

- Static biped or building: this is just a solid mesh and might be triggered only when it is clicked or touched.
- Regional mobile biped: They are in most cases NPCs that are only active (functioning), when player enters the region they belong to. Note that, this region is not the view frustum. But in special cases, view frustum can be used to activate them. So they are two sub-types of regions. Re-spawning creatures most suit this type, provided there is only one player in the map. See the figure. Currently, the engine does not support this type of biped.

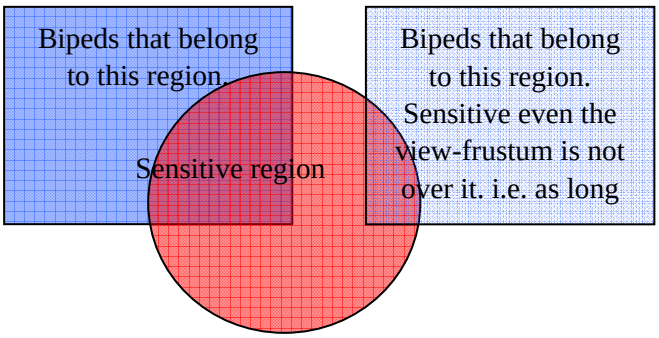

Figure 5.4 biped type

- Global biped: PCs and OPCs and some special NPCs are of this type. They are active on the map, regardless of their positions. Global biped can also be regional in the sense that they are reactive only to stimuli in a certain region. However, its script is running all the time and it is the biped who is keeping this logic, not the engine.

A simple unique structure can be used to present all three types of biped objects. All biped objects are treated like ordinary objects that are attached to Terrain tiles. Static biped are attached to the Solid object list with all other kind of mesh objects in the CTerrainTile class instance. Regional biped are attached to biped list of the proper tile in which they are sensitive. And global object are attached to the root Terrain Tile's biped list in which they are always sensitive. The player object can be any objects that are in the list of biped attached to the root terrain tile.

Collision detection

First of all, the game engine only generates collision pairs for active objects that are in the simulation region. Second, some active objects usually reside nearer to the scene root than they should be, hence we must test against all other active objects in the same tree level as well as the deeper part of that tree node (which might not be transversed in rendering pipeline). An outline of the algorithm is given in Table 5.2. (Note: Biped means a mobile object in the scene graph.)

Table 5.2 Environment Simulator: Collision detection

Clear the global tile list, biped lists, and visiting biped list in each active tile
Pass1: Select bipeds marked with OBJ_VOLUMN_PERCEPTIVE_RADIUS into the Active Biped List
Pass 2: generate static collision pairs for each moving active bipeds.
"static collision pair" means that one object in the pair is static i.e. not marked with
OBJ_VOLUMN_PERCEPTIVE_RADIUS.
Pass 3: add any active biped that is in the perceptive radius of each active biped into its vicinity biped list. If the biped is moving and is in collision with other active bipeds, add them into its collision list.
For each biped in each active terrain tile in the list
Test with all other bipeds in the same tile.
end
Pass 4: Animate biped, i.e. move it around the scene, executing High Level Event and/or changing its locations). Solve all collision pairs for each active biped. Path-finding is implemented next; additional way points will be created for mobile objects if necessary.
Pass 5: Execute AI modules for each active biped.

After implementing the above algorithm, we have a list of biped that is active in the current frame, the tile that contains it and any other biped object that is also in the tile. This

information is then passed to the AI module of each active biped as their environment input (stimuli).

5.3 Physics Events

When the environment simulator discovers that a mobile object such as a biped is in collision with other mobile objects or selected by mouse input, it will send a message to its associated script file, if any. The environment simulator will also generate low level event, such as to stop a biped from moving forward. Low level event is always carried out internally and the users can not intervene.

5.4 Conclusion and Ongoing Work

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, while others are approximated. Both the physics model presentation and simulation algorithms can differ greatly for different types of objects in the 3D world. For example, the global convex 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 the occasion. For example, suppose there is a cave inside the global terrain. A game engine can not decide whether the cave physics or the global terrain physics should be applied at the intersection regions. In such cases, some additional information should be given for the decision to be made correctly. In the above case, we may add a portal physics object, so that the cave physics will always precede the global terrain physics in regions covered by the portal physics object.

In ParaEngine, I have implemented physics simulation for the global terrain, the indoor buildings, outdoor meshes and some of their combinations. Some of the ongoing work may be supporting physics in deep water and on water surfaces, and allowing characters to walk through tunnels, etc.

Chapter 6
Navigating in the 3D world

In previous chapters, I have shown how a 3D world can be modeled, rendered and simulated. This chapter focuses on navigation in the 3D game world. The camera system is the primary interactive navigation tool for human users; whereas characters in the scene also need their own navigation system to travel in the world either automatically or through some predefined routes. Some techniques of autonomous character navigation are also discussed in Chapter 8.

6.1 Automatic Camera Control

In ParaEngine, constraint based [43] camera control is implemented. Related works on camera in 3D environment can be found in [44] [45]. The camera constraint that is related with the physics is the occlusion constraint, in which the camera should have an unobstructed view of some point or object in the 3D environment.

6.1.1 Background

Automatically planning camera shots in virtual 3D 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 virtual 3D 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. A human user or intelligent software system issues a request to visualize subjects of interest and specifies how each should be viewed, then a constraint solver attempts to find a solution camera shot. A camera shot can be determined by a set of constraints on objects in the scene or on the camera itself. The constraint solver then attempts to find values for each camera parameter so that the given constraints are satisfied. This paper presents a work in progress snapshot of the virtual camera constraint model that we are currently developing.

6.1.2 Occlusion Constraint

Implementation of the occlusion constraint is given as below.

Input:

- vEye: new eye position
- vLookAt: new look at position

Algorithm:

We cast a ray from the new look at position to the new eye position. If this ray hits something, let fLineOfSightLen be the distance from the intersection point to the ray origin. If not, the new eye position and look at position is adopted. Further reaction is subject to the value of fLineOfSightLen as below.

- If fLineOfSightLen is larger than the line of sight with the new camera position, then the new position is 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 new camera position, but larger than the distance from the near camera plane to the camera eye, the new 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

Adjusting for view frustum

For all kinds of cameras, we ensure that the near plane camera rectangular and the eye position are well above the global terrain surface. This is done by ensuring that all the five points (four for the near plane, one for the camera eye) has height (or Y component) at least larger than the global terrain at their (x, z) location. A similar method is used to avoid near plane camera intersecting with the physics 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 value calculated during the previous step.

Figure 6.1 shows an indoor camera automatically adjusted to provide an unobstructed view of the player in control.

Figure 6.1 Camera Occlusion Constraint and Physics

6.2 Generic Path Finding Algorithms

Path finding is one of the most visible types of Artificial Intelligence 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 [46] are popular algorithms, which can efficiently build optimal paths between two endpoints, even if we have to cross many miles and obstacles in the process.

Two useful path findings algorithms in games are (1) simple wall sliding and (2) A* algorithm. Please refer to other references for their implementation. I have already implemented wall sliding in Chapter 5.1.6.1. A* path finding is not suitable for outdoor games with extensible maps. And I did not implement it in ParaEngine.

6.3 Object Level Path Finding in ParaEngine

Object level path-finding is path-finding in the 3D world without taking the shape of objects in to consideration. It is mostly used for NPC character navigations. In ParaEngine, object level path-finding will use the intermediate result generated during collision detection. Path-finding is implemented by each individual biped object. The input information that a biped has is the terrain tile that it belongs to and the distance to all other bipeds and obstacles in its perceptive radius. This information is generated by the environment simulator in an earlier stage. The job of path-finding is to generate additional waypoints to the destination. There are several kinds of waypoints that a path-finding biped could generate as a result. See Table 6.1.

Table 6.1 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 rule is given in Table 6.2.

Table 6.2 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. The following steps are used to generate waypoints in path-finding solutions.

Step1: If the biped has already reached a waypoint, then remove it and go on to the next one in the queue. When waypoint itself is in collision with other static objects, the space occupied by these static objects will be used as the destination point; whereas in collision free waypoint, a destination is just a point in real coordinate system.

Step2: Check if there are other moving bipeds in its collision list. If so, block the current biped for some seconds

or if the destination point collides with any of them, remove the way point.

Step3: Get the biggest non-mobile object in the collision list. We can give a precise solution. according to its shape, when there is only one object. Any solution should guarantee that the biped is approaching the destination point, so that a group of solutions are guaranteed to reach their goals within limited time.

The core of the algorithm is this: when several objects already collide into each other, only one object is picked to implement path-finding, while the others are put into a blocked state. This moving object will then pick out the biggest object that is currently in collision with it and try to side-step it by generating additional waypoints between its current location and the old destination. This is a fast path-finding algorithm in unbounded real-coordinate system, and is compromised between efficiency and accuracy.

6.4 Path-finding in ParaEngine

In the ParaEngine, path-finding is implemented in the environment simulator. Actually there are two places where the path module can be reasonably located. One is the environment simulator and the other is AI module. Of course, the first one insures that object do not collide into one another through passively applying physical law to them. The second one does this actively by letting the intelligent object trying not to violate the law. The best method is to do both. However, for performance reasons, we will do them in the environment simulator, which clearly accentuate the importance of passively regulate solid object movement. This may create unnatural path of an object, but the result is safe and legal. I try to minimize this effect by dynamically generate some way-point during path-finding, so that the articulated path result will appear to be planned beforehand by an intelligent object with eye sensors. See Figure 6.2**.**

It is extremely difficulty to implement path-finding algorithm in a real-number coordinate system, where there is no restriction of where and how solid objects are placed in the world, and no constraint to the position of the player. That is to say, the player can be at any position in the set {x,y,z}, where x,y,z are real numbers. Collision therefore can not be defined by 0, 1 two states, but by a function DepthOf (object1, object2) whose domain is also real number set. The larger value is, the deeper the two objects run into each other. Such a path-finding algorithm in real-number coordinate system is to find the next position, where the newly computed depth value for every two objects in the scene decreases. This can be a difficult task when the object's position and movement has no restrictions. Graph-based path-finding algorithm (like A*) can not be applied directly.

To see how I solve this problem, please refer to (ParaEngine, 1.1). The basic idea is to generate waypoints (real number tuple {x,y}) dynamically from nearby objects. Waypoints are generally vertices of box object or points in the tangent line of a spherical object. And then find a solution from among the constructed waypoint list.

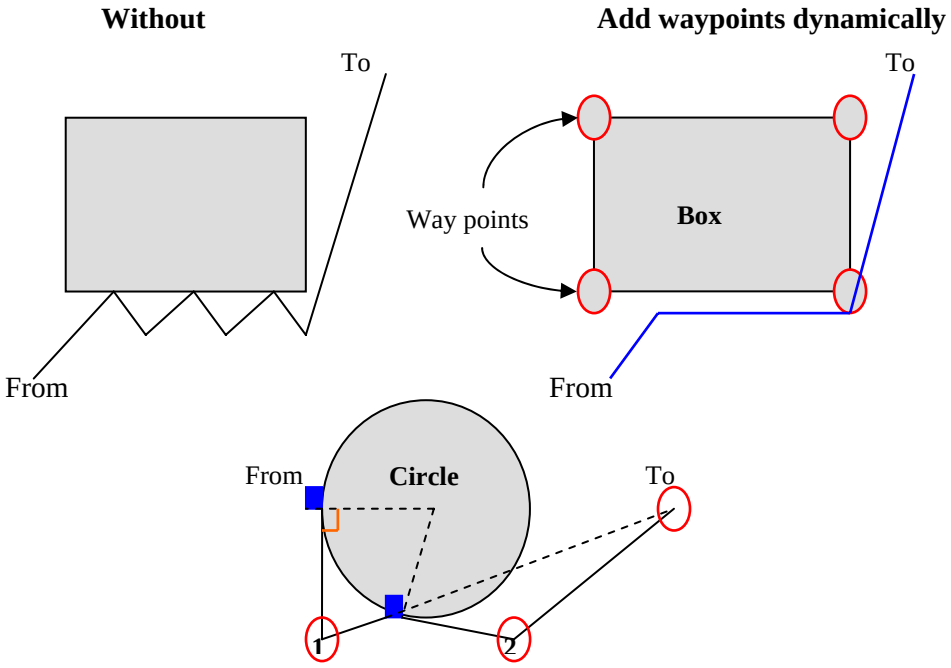

Figure 6.2 path-finding: adding dynamic waypoints

6.5 Conclusion and Ongoing Works

In distributed game world, there are three levels of navigation.

- 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.

Currently this chapter only covers navigation in the first two categories. The last one is my ongoing work.

Chapter 7
NPL Scripting System

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 web3d [2], 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.

7.1 Motivation

In my viewpoint, the compiling of code (that targets distributed environment) may also be carried out in a distributed manner (from command-line compiler to rich HCI enabled ones with network capabilities); the next generation high-level language may be able to express adaptive and distributed behaviors with its own language primitives; its compiler may be able to generate low-level code that runs on any part of the network; and its development environment may allow visualized design of any parallel-code and deployment-scheme. In other words, the coding and compiling process may both be carried out in a distributed manner and environment. This calls for a new language dedicated to this task and a new human-computer interface (HCI) adopted by its compiler and runtime environment.

With this vision, I proposed a unified approach of a neural network based programming paradigm called Neural Parallel Language (NPL). Distributed software systems generally need to solve two problems: computing and visualization. NPL is associated with a game engine which provides a visualization platform for the language.

7.2 The NPL methodology

The key idea of NPL is that software system in the future functions more like one giant brain spanning across the entire Internet. It could form new neuron connections, learn from experiences, remember patterns and perform many other functions resembling the human brain. Current object-oriented programming language lacks the directness in composing such kind of software systems, nor is any existing Neural Network Simulation Language eligible for constructing commercial distributed software. NPL tries to solve these problems by means of (1) keeping all communication, network deployment and certification details out of the

program code, (2) presenting programmers a very clean neural network based programming paradigm, (3) preserving all previous familiar paradigms such as object oriented or functional programming. By using NPL, software is constructed like designing a brain network. Section 2.3 shows a demonstration.

The components of NPL include:

- Neuron file: The source file that programmers used to code the function of an abstract neuron. NPL does not distinguish between a single neuron and a network of neurons. Both can be modeled inside one neuron file. No explicit instantiation is needed in the code, so long as files are deployed into a runtime environment. More details will be given later.
- Runtime environment: It is a management environment where activation and execution of local neuron files occur and messages to external neurons are routed via network. It maps resource names (e.g. of neuron files) to their physical locations on the network, and automatically update any topological changes to this mapping.
- Visual compiler: It compiles neuron files into intermediate code to be executed by the Runtime. It has a GUI front end to allow a group of neuron files to be deployed (compiled) to multiple Runtimes on the network.
- Visualization and simulation Engine: It provides a complete class of multimedia functions (Host API) which neuron files can be used to read/write to/from a networked simulation environment. This can be regarded as a shared space of virtual reality theatre where stories are evoked or conjured up by a neural network. This virtual reality space may also generate stimuli back to the neuron files. We have and will continue call such a simulation environment a game engine in this paper.

7.3 Distributed Visualization Support

In a typical scripting system, logics can be embedded in script files and reused by activating each other; script states are saved in global tables of a script file; objects in global tables can also be accessed by other scripts and are usually served as an IO pool for data exchange between different scripts.

As mentioned previously, I have extended the above local scripting environment to networked scripting environment. It is achieved by regarding all networked computing resources (script files) as possessed by one big super computer, so that all concepts that worked on a single runtime will also work for the networked runtimes, except for one thing: unlike on a super computer, script files on the network no longer serve a single user. Instead, some script files may need to process inputs from several game engine instances simultaneously (i.e. in the same discrete time step) and emit visualizations to the corresponding game engine instance during its activation chain. If we reuse the analogy to the human brain in Chapter 2.1, the same unconscious mind will need to handle multiple perceptions and attentions.

Due to the confusing relationship between computations in script files and their visualizations, the following two approaches have been taken in other virtual reality (VR) systems.

First: scripts from another runtimes are duplicated in the local runtime. In fact, all HTML pages on the Internet use this simple approach (i.e. they are downloaded to your internet explorer on demand for display.). However, it is both time consuming and unsafe to synchronize local copies of these scripts when applied to the game scripting context.

Second: Fixed view-distribution architecture is enforced in some script files, such as a server script providing visualizations for many view clients. In fact, some server-side scripting on the Internet, such as ASP and PHP uses this approach. However, in the game scripting context, it will lose much of the flexibility of scripting as provided by a single runtime.

The ideal solution of scripting runtime for distributed games would be a type of visualization architecture where (1) each atomic script file can emit visualization commands anytime and anywhere during its execution; (2) these visualization commands wisely know to which game engine instance they go to, unless the script has the necessity to know all the game engine instances for which it has served.

For example, in our game demo, a player might simultaneously interact with several objects in a game scene. Their events will activate the corresponding scripts on several remote servers, which in turn may further activate other scripts on other severs. All scripts along the activation chain may issue visualization commands to change the 3D scene on the original player's computer. A more concrete example is that one player is collided with another player in a game scene. What happens to the first player's runtime environment? One possible computing scenario behind it is stated below. The first player's runtime sends a message to the terrain host server, whose scripts will generate GUI commands to update player locations in the caller's world. Meanwhile, another message is sent to the second player's server. Scripts on this second server might generate a GUI command to display some text on the first player's world. The aggregated world changes for the first player are that the positions of all visible players are updated and that a piece of text is displayed in a popup dialog box. From the first player's viewpoint, the computing occurs on remote servers; GUI commands are issued from remote scripts, and then interpreted and executed in its local game engine instance.

7.4 Message Driven Model

Many interpreted extension languages are event driven. However, a 100% event-driven language can not simulate parallel behaviors, unless it's been explicitly programmed as multi-threaded. This is because functions or nested functions must be fully executed before it can release control of its execution thread. Another extreme is that functions can be suspended at any point of execution, at the cost of maintaining mutual exclusive access of any shared data structures. For functional and performance concerns, none of these methods is used by NPL message driven model. Instead, NPL runtime environment adopts a hybrid approach. It divides time into small slices. Within each slice, there are two phases (see Figure 7) called (1) synapse data relay phase, (2) neuron response phase. In phase 1, stimuli from the environment, network and/or local neurons activate the synapses of any connected neuron; and data is passed to the soma, cached, but not executed. In phase 2, NPL examines the list of potential reacting neurons, which is generated in phase 1, and executes it if any of its input field condition tests has passed. Any single execution should be guaranteed to exit within the rest of the time slice by the programmer. The executable code may include (a) further activation

test, (b) generating stimuli to some other neurons wherever they may be, (c) calling Host API functions provided by the host application (game engine).

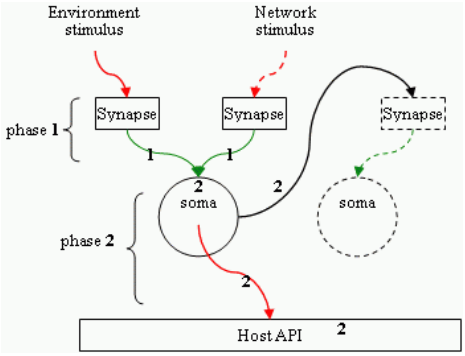

Figure 7.1 Time slice and phases

With this simple approach, there are three obvious advantages. (1) The time interval during which the neuron state is changed is fully predictable (it is always in phase 1). NPL can efficiently handle mutual exclusive data access to any input field data. (2) External stimuli (from network) are handled transparently as internal stimuli. (3) The execution of NPL never stalls the CPU; even it is running in the same thread as the host application (game engine). This feature also makes it easy for the neural network to communicate with the game engine, because by running in the same thread, it automatically guarantees mutual exclusive data access to the engine and vice versa. The current implementation of NPL only supports this single threaded mode.

In computer game engine (or other discrete time interactive applications), one annoying problem is that when dealing with some computation intensive tasks, the graphic (or other real-time function) jerks. NPL generally solves this problem, because execution can always be paused at predictable short interval to free CPU for graphic rendering or IO polling.

7.5 Neuron File

In NPL, neuron file plays an important role. It is the building block of neural network. Each neuron file represents an abstract neuron and can have one activation function and many helper functions. Files are referenced by other files through namespace shortcut. File is used to represent NPL neuron for the following reasons. (1) File is automatically managed in most operating systems; and people are familiar with it. (2) The deployment and configuration of files on the network is easy; and it is supported by operating system. (3) File is the most common Internet resources. Ontology can be created for a domain of neuron files on the network. (4) By using a single file, NPL gets rid of any artificial tokens and syntax that may bewilder programmers at first. E.g. a blank file in NPL is also a valid neuron file and is able to receive and store (overwrite only) any incoming signals, but not producing anything. (5) By setting default variable scope to global (of the file), the states of a neuron file can be easily managed.

Object oriented programming is allowed in any place of the neuron file. E.g. the string data type is an internal object that contains both data and functions and can have many instantiations in different places. However, each neuron file is immediately an instance of itself so long as it has been assigned a path name in a namespace, which is done by CARE.

All other neuron files can begin referencing this neuron instance by this unique name or its shortcut. The NPL runtimes maintain a mapping from such names to their physical addresses during execution.

7.6 NPL Network Ontology

Neuron, neuron input/output field, neural network can all be assets on the Internet. RDF is an ideal framework of expressing such ontology, and neural network could be made universally available. Discovering and generating ontology might be a joint job of CARE and the human user. With an ontology framework, (1) Neuron files could be referenced by namespace shortcut (i.e. shorter and invariable names) rather than physical addresses (2) it allows the runtime environment to quickly update network topological changes inside a domain of neural network. (3) it enables two unknown neural networks to connect to each other so long as they have agreed upon some IO rules defined in some global files.

Currently I did not implement the ontology approach for locating neuron files (it is an ongoing work). For simplicity, the current NPL runtime searches the file directory for a configuration file of any dummy neuron (a neuron file that does not exist on the local computer). It begins from the current directory, then the parent, etc. This is a distributed approach, but is not very convenient. For example, files belonging to the same namespace must be deployed to a file directory with the same short name as the namespace on all runtimes (computers).

7.7 NPL Scripting System Overview

In distributed game engine, it is common to deal with thousands of interactive entities on the local runtime and among its web servers. Our solution is based on scripting technology, which has already been broadly used in modern computer game engine to provide flexible and extensible integration of a single game engine runtime and the rest of the world.

Scripting system is a high-level language and runtime environment which sits between the core of a game engine and the human beings as well as intelligent agents. Briefly speaking, the following workflow applies in some game engine configurations. For each game engine instance on the network, it will (1) update changed game states from input sources, (2) simulate game world objects which contain local physical properties, (3) generate physics events and other feedbacks to intelligent entities for decision making, and (4) render the portion of the world within the camera view to the display. In (1), a large portion of game engine input (with the exemption of direct scene manipulation) comes from the scripting system. In (3), most events and feedbacks (perhaps only with the exemption of local haptic devices) are reported to the scripting system runtime.

The scripting system consists of a collection of activation files as well as other reference files. An activation file may be activated by other script files, some synchronization timers and the game engines. Logics are usually scripted (written) in activation files and carried out by series of file activations, during which process activated files may emit UI commands to game engines for game state updates. The mechanism greatly resembles an artificial neural network, where the game engine plays the role of our internal visual and auditory perceptions.

In fact, it is worth mentioning here that a theory or hypothesis of the human brain concerning Imagery [9] [10] has been studied long ago. Metaphorically speaking, it compares

human Imagination to a multimedia, virtual reality “theatre”, where stories about the body and the self are played out unconsciously and an attention mechanism selects the most relevant simulations to our inner perception and be carried out by action. In comparison, game engine carries out some low-level simulation and also provides hints to the attention mechanism (its camera system, perhaps); the scripting system does all sorts of unconscious computing and emits bits of visualization results to the game engine for the selection by attention.

Game engine practitioners have used some of the aforementioned scripting framework to add soft computing capabilities to a variety of their engine modules. We have extended the scripting system to make it suitable to use in distributed game world composing. Instead of restricting scripting system to a local environment, our approach is to allow arbitrary distribution of activation files on the network and use global naming conventions to reference them. The script system automatically manages all network connections, so that the activation chain of script files may span an arbitrary network and link any number of game engine instances transparently.

7.8 Distributed Visualization Framework Design

Each distributed game may define their own set of virtual world constructing functions (or visualization commands as used in previous text). In practice, this is usually done through extending an existing script language such as Lua [47] or Python. A script (or activation) file should at least contain a function body called *activate* (code). This is like the main () function in C programs. Once a script is activated either by other scripts or by the local game engine, the *activate* function will be called by the script runtime. All script language must also support an asynchronous function called *NPL.activate* (sScriptPath, sCode) which takes two explicit parameters. The first parameter is a path name of the script file which will be activated; the second parameter is a program code to be executed in the global space of the specified script file. The sCode parameter may be used to transmit data to another script. The function will return immediately, and its specified script will be activated in the next time step or if the specified script is on a remote computer, the script runtime will automatically send it via available connection or establish a new one. A sample script code is given below.

```
function activate(code)
  if (state == nil) then
    … …
    local player = ParaScene.CreateCharacter ("Actor",    "biped mesh", "", true, 0.5, 0, 1.0);
    … …
    ParaCamera.FollowObject(player);
    -- activate another script.
    NPL.activate("ABC: \\pol_intro.npl", "state=nil;");
  else
      … …
```

```
   end
end
function Service1()
   … …
end
global items[]; -- a global variable (table)
```

In the above code, Para*() functions are game specific UI functions (or visualization commands). These UI functions cover a complete set of game content controllers, such as loading game scene objects, displaying dialog boxes, playing sounds, updating player positions, animating characters, changing camera mode, etc. In the *NPL.activate* function, the file path parameter contains the string "ABC:" which is a namespace shortcut that will be converted to an actual network address by the script runtime during execution. A script file is also allowed to contain any number of global functions called services, such as Service1 () in the sample. Services and all global variables are accessible by other script files via the sCode parameter of the *NPL.activate* function, whereas everything inside the *activate* function are private to the script file.

Two major problems in distributed scripting systems are (1) data sharing (2) data visualization. The first problem can be solved by extending the addressing space from local to global (as already shown) and passing any certification information along with data access requests (which is automatically done by script runtimes). However, the second problem is not trivial and does not have a universal solution as the first one. As discussed in Section 4.1, we want an ideal solution to automatically emit visualization commands to the right game engine instance. For example, in the above code sample, Para*() UI commands should be automatically emitted to the right game engine instance, which may or may not be the local game engine instance.

To achieve this, the script runtime maintains two implicit lists for each activated script file. One is called UI functions list; the other is called UIReceivers list. The latter contains a collection of game engine instance addresses. During the execution of a script file, the script runtime will store any encountered (emitted) UI functions (replacing parameter names with actual values) to the UI functions list. Likewise, for each encountered *NPL.activate* function and UI function, it will attach an instance of the current UIReceivers list to it and proceed. Consequently, in the viewpoint of the scripting runtime, the *NPL.activate* and all UI functions have an invisible parameter called UIReceivers list. Finally, at the end of an execution step, the script runtime will sort all UI functions in each script files by their UIReceiver list and batch-send UI functions to game engine instances whose address appeared in the UI function's UIReceiver list. As for the *NPL.activate* function, the UIReceiver list is passed on to the specified script file and becomes that script file's initial UIReceiver list when it is activated.

So far, we have shown how UIReceiver list is passed on during a script file activation chain. The content of UIReceiver list can be filled up or changed in three places: first, a chain of script file activation usually originate from a game engine event such as a collision event; the address of the triggering game engine instance is automatically added to the UIReceiver list which is passed to the script file. Second, if a script is neither activated by other scripts nor by

a game engine event, the local address is added to its UIReceiver list. Third, UIReceiver list can be accessed and modified in the script file itself via a special function called *NPL.GetUIReceivers()*.

Figure 7.2 UIReceiver sample traces the UIReceiver changes during an activation chain which originates from game engine instance A. The result of this activation chain is that UI functions UI_1(), UI_2() and UI_3() are executed in game engine instance A and UI_3() is executed in game engine instance B.

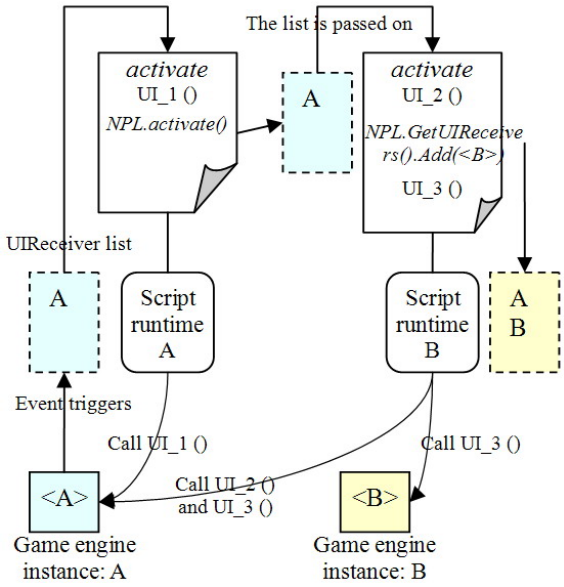

Figure 7.2 UIReceiver sample

The UIReceiver mechanism can be completely transparent to script files so that each script file does not have to worry about to which game engine instance its UI functions should be sent. For example, we can write a script file which contains logics of an intelligent agent in a 3D game world. Suppose there are a dozen visitors from different game engine instance talking to this agent, the agent's script file will be activated a dozen times, each time a different address is stored in its UIReceiver list. Hence, the same script code is executed, while multiple game engine instances are served. Of course, if the agent really needs to care about which game engine instance it is currently serving, it can still access such information through NPL.GetUIReceivers() and respond accordingly. The same mechanism also works for longer and much complex activation chains with or without feedbacks.

7.9 NPL Runtime Architecture

The original design and implementation of NPL can be found in (ParaEngine, 2.3, 2.4). Currently my friend (WangTian) and I have designed a new and more efficient implementation based on the distributed visualization framework proposed in previous sections. But the implementation has not been finished. So I will only include some early architecture of the new NPL implementation.

This section shows NPL runtime environment implementation. NPL runtime itself is embedded in each local game engine and manages communication between neuron files.

Figure 7.3 shows NPL runtimes on a computer networks. Each node in the figure denotes a single computer.

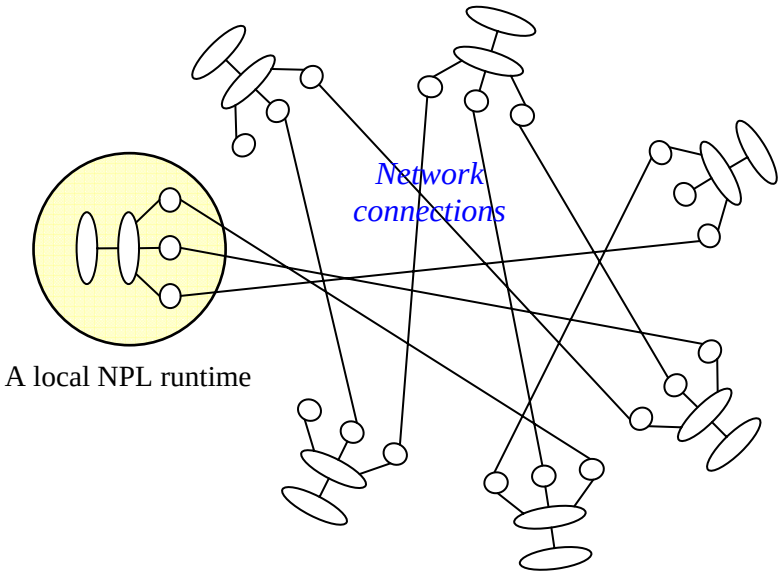

Figure 7.3 NPL Runtimes on a computer network

7.9.1 Architecture Design Goals

The design goal of NPL runtimes is that although there are many NPL runtimes on the network, the programming and communication interface act as if there is only a single runtime environment for all script files on the network.

7.9.2 Architecture Design

For each local runtime in Figure 7.3, Figure 7.4 gives its inside architecture. Each NPL runtime is both a client and server. A connection between two different NPL runtimes on the network is established through peer-to-peer sensor objects.

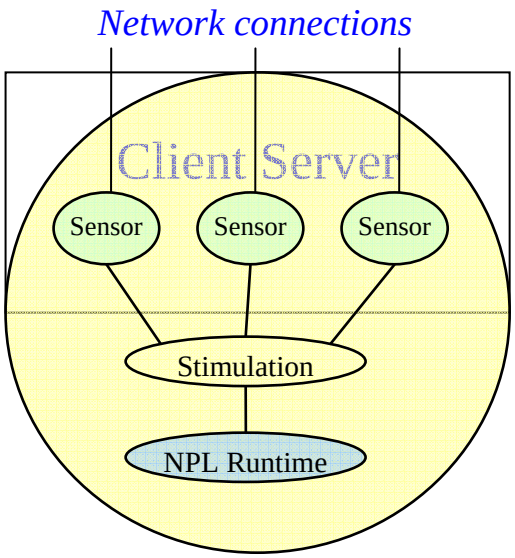

Figure 7.4 Inside NPL Runtime

A pair of sensors exchanges information in the form of packages. Figure 7.5 shows the package specifications.

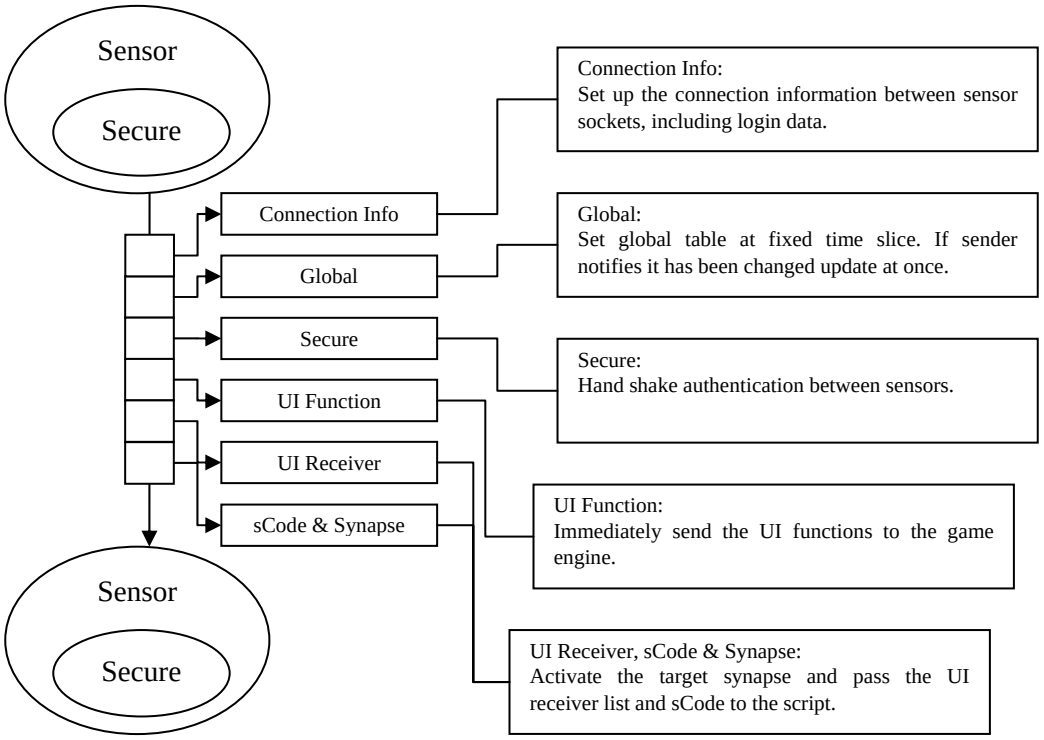

Figure 7.5 Packages between sensors

7.10 Composing Distributed Game World Logics

The following is the basic steps of composing a game world with current available tools in ParaEngine.

Step1: Preparing game assets. These include mesh models, biped animation sequences, sounds and textures. Or one can collect URI of such models if they are Internet resources. Instead of using URI or file path name for their instantiation or elicitation later in scripts or VRML file, they are given shorter names. This is done by a short script. A sample code is given below:

```
function wrl_movie1_res()
    -- X file terrain 200*200
    ParaCreateAsset("MS", "terrain200", "xmodels\\terrainPH.x");
    --anim:[0]stand,[1]stand hit,[2]death,[3]Birth,[4]Spell
    --radius:1.346700 meters
    ParaCreateAsset("MA", "tree0", "Doodads\\Terrain\\AshenTree\\AshenTree3.mdx");
end
```

A GUI tool has been created for exporting groups of resource files into such script code. The above script is actually generated by this tool. Functions started with “Para” are Host APIs. Therefore, by referring to this script, all runtimes on the network know where to find these named resources.

Step2: Building 3D scenes. 3D Scenes are built using 3dsMax and exported as a VRML file. These scenes (files) are visual entities that might be called by the neural network as visual elicitations. They will cause the game engine to present its imagery to the computer screen; in the meantime, the engine simulates the imagery which might cause new stimuli to be generated to the neural network. Hence this forms the imagery-subconscious loop previously mentioned in Figure 2.1. In our current implementation, VRML file needs to be further compiled to a more compact script format by a cross-compiler tool.

Step3: Constructing Neural Networks using NPL. This is the most important and high-level part of distributed game world composing. In our framework, neural network defines the behavior of the game world and its logics. For example, one can create NPL neuron-file network that functions as message broad-casting portals, reactive agent (like RPC or remote procedure call), memory block, a sequence of cinematic, complex logic circuits with feedbacks, or a router or switcher, etc. Figure 7.6 shows a most simple demo: building a movie clip or cinematic with a NPL neural network.

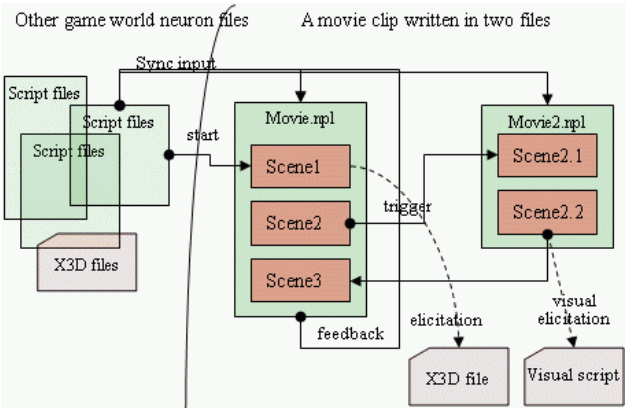

Figure 7.6 NPL demo: a simple cinematic

This network contains only two neuron files: Movie.npl and Movie2.npl. The small block inside the file box denotes a kind of input activation conditions. Each activation condition is symbolically named as SceneX. The sync input field is one method of synchronizing the beat (activation) of the two neuron files. The execution of the cinematic is rather like real movie shooting. Each SceneX will either elicit a complete visual imagery (by referring to an X3D file or visual scripts) or call Host API functions (of the game engine) to reset the camera or assign tasks to scene objects.

Step4: Deploying neuron files on the physical network. Neural network deployment shows one benefit of using neural network based programming paradigm in distributed application development such as composing game world logic on the Internet. Neuron files can be arbitrarily distributed on the physical network. For example, in Step3, the two movie files can be deployed in one or two computers. When composing a game world, designers can usually divide its logic into two general categories: client side and server side. Client side neuron files will be shipped with the game; while server side will be distributed to many host servers. An alternative might be regarding each computer (peer) as both client and server. Information unspecified in the neuron source code (such as the topology of neuron files) is dealt with by CARE of the NPL language.

7.11 Conclusion

Scripting is a useful technique in virtual game world constructing. Traditionally, it has been used to separate the low level game programming from the high level programming. In this literature, I have exploited another possibility of using script system to construct distributed logics on the network. Scripting system provides an interface not only between the game engine and human users but also between game engines and intelligent agents. I hope that the proposed scripting system will accelerate this process by making logics on the network easy to write and deploy.

Chapter 8
AI in Game World

In the previous chapters, I have discussed the general framework and implementation of a distributed game engine. In this chapter, I will focus on a new aspect of the game engine, which is artificial intelligence (AI) in game world. Although the topic of AI is more specific to games than to game engines, there are some common paradigms that most game specific AI follows.

8.1 NPC: Reactive Agent

This section is not implemented. But in my game demo, I use a simple finite state machine with a neuron script file to model some NPL characters that could talk to players with interactive menus.

8.2 Re-spawning Creatures: Simple Active Agent

This section is not implemented. But I have written some simple AI controller, which could be assigned to biped objects through scene object event. See below Table 8.1. But they are no longer used in the recent release of ParaEngine.

Table 8.1 Some AI controller events

script	Description
`ParaAddEventToObject("enemy1", "asai", 1, 1);`	Assign ranged creature AI module to enemy1
`ParaAddEventToObject("friend1", "asai", 1, 0);`	Assign melee creature AI module to friend1
`ParaAddEventToObject(nil, "colr", 1, 0, 0);`	Assign red color to current obj(nil)
`ParaAddEventToObject(nil, "task", "Follow", "Li");`	Follow “Li”
`ParaAddEventToObject(“friend1”, "task", "Evade", "enemy");`	Evade “enemy”
`ParaAddEventToObject(nil, “task”, “Movie”, “<Attack,50,0,60,0.6,4><Die,50,0,60,0.6,4><Decay,50,0,60,0.6,4>”);`	Play movie with key frames: <const char* anim, float x, float y, float z, float facing, float duration>. If y=0, then the height of the position is ignored
`ParaAddEventToObject("nil", "anim", "Attack");`	Play animation “Attack”
`ParaAddEventToObject(nil, "anim", 5);`	Play animation number 5

Figure 8.1 shows a group of NPC fighting against each other. These NPC are all controlled by the above AI controllers.

Figure 8.1 Characters driven by AI controllers

8.3 Player: A Passive Object

This section is not implemented.

8.4 Autonomous Character Animation

In Chapter 2, I have proposed a simulation framework. In this section, I will extend that framework and use it in autonomous character animation in the game engine.

8.4.1 Introduction

Real-time human animation covers a wide range of applications such as virtual reality, video games, web avatars, etc. When animating a character, there are three kinds of animation 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 paper mainly concerns about 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 and sounds).

The motion of a specified human character can be formulated by a set of independent functions of time, i.e. $\{f_n(t) \mid n = 1, 2, ..., N; t \in [T_0, +\infty]\}$. 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 $\{f_n(t) \mid n = 1, 2, ..., N; t \in [T_0, T_c]\}$ and the environment W (or workspace in robotics), computing $\{f_n(t) \mid n = 1, 2, ..., N; t \in [T_c, T_c + \Delta T]\}$ 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 figures [48] [49]. There are also several sophisticated combined methods [50] 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.

This paper presents a different motion generation framework aimed 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. In section 2, we will introduce the theoretical basis for this motion generation system. In section 3, we will give the implementation suggestion from its realization in a computer game engine we developed. In section 4, we will discuss how the research fits into the large context of automatic motion synthesis in the computer game engine which includes both global and local animations.

8.4.2 Mathematical and Architectural Formulation

To make things more precise, we now give a more formal formulation of the motion synthesis problem for animated characters.

8.4.2.1 Formulation of the Physical World

The notation adopted here is loosely based on the conventions used in [51].

1. The 3D environment in which the characters move is denoted by *W* (commonly called the workspace in robotics), and is modeled as the Euclidean space R^3 (*R* is the set of real numbers).
2. All local environmental variables are denoted by $\mathbf{e} \in E$, a vector of *m* real numbers, specifying the environmental or emotional states relevant to a certain character.
3. A character or agent is called *A*. If there are several characters, they are called *Ai (i = 1, 2, …)*
4. Each character *A* is a collection of p links *Lj (j = 1, … , p)* organized in a kinematic

hierarchy with Cartesian frames *Fj* attached to each link.

5. A configuration or pose of a character is denoted by the set *P = {T1, T2, … ,Tp}* of *p* relative transformations for each of the links *Lj* as defined by the frame *Fj* relative to its parent link's frame. The base or root link transformation T1 is defined relative to some world Cartesian frame *Fworld*.
6. Let *n* denote the number of generalized coordinates or degrees of freedom (*DOFs*) of *A*. Note that *n* is in general not equal to p. For example, a simplified human arm may consist of three links (upper arm, forearm, hand) and three joints (shoulder, elbow, wrist) but have seven *DOFs (p = 3, n = 7)*. Here, the shoulder and the wrist are typically modeled as having three rotational *DOFs* each, and the elbow as having one rotational *DOF*, yielding a total of seven *DOFs*.
7. A configuration of a character is denoted by $\mathbf{q} \in C$, a vector of *n* real numbers, specifying values for each of the generalized coordinates.
8. Let *C* be the configuration space or *C-space* of the character *A*. *C* is a space of dimension *n*.
9. Let *Forward(**q**)* be a forward kinematics function mapping values of *q* to a particular pose *P*. *Forward(**q**)* can be used to compute the global transformation *Gj* of a given link frame *Fj* relative to the world frame *Fworld*.
10. Let *Inverse(P)* be a set of inverse kinematics (IK) algorithms which maps a given global transformation *Gj* for a link frame *Fj* to a set *Q* of values for **q**. Each configuration $\mathbf{q} \in Q$, represents a valid inverse kinematic solution (*Forward(**q**)* positions the link *Lj*, such that the frame *Fj* has a global transformation of *Gj* relative to Fworld). Note that the set *Q* may possibly be infinite, or the empty set (no valid solutions exist).

8.4.2.2 Formulation of the Motion Control System

Any motion for the character (including the changes in its local environment) will trace out a curve (i.e. a path) in this multi-dimensional space as illustrated in Figure 8.2. The curve is normally a piecewise continuous function of time (i.e. a trajectory). Conceptually, the fundamental goal of the motion synthesis strategy is to generate trajectories in the configuration space, so that they are in harmony (e.g. trajectories of the environment variables match the desired trajectories of the animation variables.). The character will either act out its planned motion or the one dictated by a supervisor. In whichever ways, the character will observe its final actions and always consider them to be in harmony. The continuous observation provides the basis for the generation of its future harmonious motions.

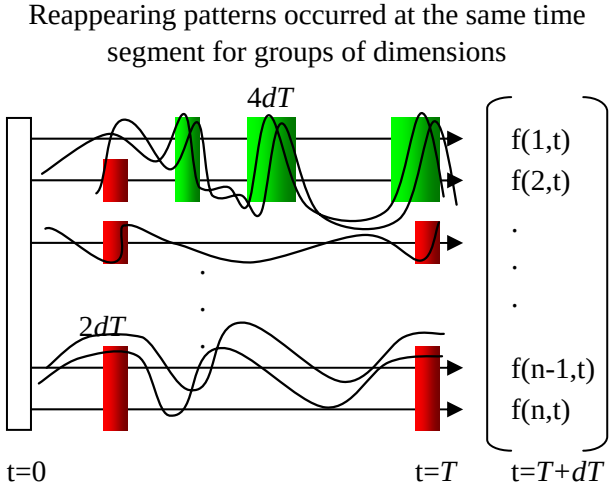

Figure 8.2 Motions for a character trace out a time-parameterized curve in the multi dimensional space.

1. A snapshot of a motion at time *t* is denoted by **f** (t) = $<f_i(t)> \in D$, where $D = E \bigcup C$. **f** (t) is a vector of $N = (m+n)$ real numbers. The vector space *D* is called DIM (dimension) space which combines the configuration of the environment variables and that of the animation variables.
2. A *simulation* is defined as *sim*(**n**, *t1*, *t2*) = $\{f_{n_i}(t), t \in [t1,t2] | n_i \in \mathbf{n}\}$, where $\mathbf{n} \subseteq \{i \mid 1 \le i \le N, i \in N\}$
3. Let **ALL** *(t)* be a set of *N* functions of time *t*, also defined by *sim*({1,2,…,*N*}, *0*, *t*). Let *f (i, t) :=* $f_i(t)$ denote the *i*th dimension or componet of **ALL** *(t)*.

8.4.2.3 Motion Generation Algorithm

In this section, we will propose the motion generation algorithm. A motion system is denoted by **motion** (*Tc*) := < *Tc*, **ALL** *(t)*, **attention**, **relation**, **User** >. Let **MoGen** denote the motion generation algorithm so that **motion** ($t+\Delta t$) = **MoGen (motion** (*t*)).

Tc is the current time, where *Tc*>=0.

ALL *(t)* has the same definition as in Section 2.3.2, where 0<=*t*<= *Tc*.

attention is a function mapping each component of **ALL** *(t)* to a value att **(t)**, it is also written as **att** (t):= $\{att_i(t) | \forall f_i(t) \in \mathbf{ALL}(t)(att_i(t)=attention(f_i(t)))\}$

relation is an *N* dimensional matrix. The value **R**i,j = **relation**(i,j) denotes the degree of relativity between *f (i, t)* and *f (j, t)* for all *t*.

User is an external supervisor. It has the right to override values in **ALL** *(t)* for all *t*. **User** is an unknown process to the motion system, but is executed at the end of every time slice.

motion ($t+\Delta t$) is computed from **motion** (*t*), using the following iterative process.

(1) $f_i(T_c+\Delta T) = f_i(T+\Delta T)$, where T satisfies $att_i(T) = \max\{att_i(t) \mid 0 \le t \le (T_c-\Delta T)\}$ if no external input from the **User**; otherwise $f_i(T_c+\Delta T) = User_i(T_c+\Delta T)$. For $t \in (T_c, T_c+\Delta T)$, an interpolation function can be used such as $f_i(t) = (f_i(T_c+\Delta T) - f_i(T_c))/\Delta T \times (t-T_c) + f_i(T_c)$. After this step, **ALL** *(t)* of **motion** ($Tc+\Delta t$) can be obtained.

(2) Compute **att** (*t*) of **motion** ($Tc+\Delta t$) from **att** (*t*) of **motion** (*Tc*) and **ALL** *(t)* of **motion** ($Tc+\Delta t$), using Simulation-theory(ST rules). See Figure 8.3 (**att** (t) denotes **att** (*t*) of **motion** (t)*)*.

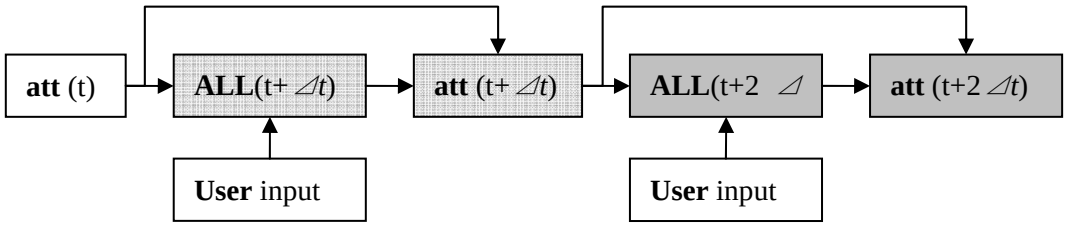

Figure 8.3 Iterative process of the motion generation algorithm

Simulation-theory (ST rules) is the center of the algorithm. It is used to calculate the redistribution of **attention** for **ALL** (*t*). To express the ST rules more clearly, the encoding of **ALL** (*t*) is given and a few other related helper functions are also defined.

ALL (*t*) is encoded using the following method. For every component f(i,t) of **ALL** (*t*), partition f(i,t) into minimum number of segments such that f(i,t) is monotonic in each segment.

$f_i(t) = \sum_{k=0}^{num-1} q_k(t - T_k)$, where $q_k(t) = f_i(t+T_k)$ if $0 \le t \le T_{k+1}$ or $q_k(t) = 0$ otherwise . $\{T_k\}$ is the set of bending points in f (*t*), where T0=0, Tnum=f(Tc+ ⊿ *t*). $\{[T_k, T_{k+1}] | k=0,1,...,num-1\}$ is the set of all segments in the time domain.

For every component y=f(i,t) of **ALL** (*t*), define its inverse functions as:

i = q(t, y), where **e**.y = f(**i**, q(t, y)) and **e** is a unit vector with the same dimension of **i**. We can use this function to get the index of components of **ALL** (*t*) which has the value of *y* at time *t*.

t = g(i, y), where **e**.y = f(i, g(i, y)) and **e** is a unit vector with the same dimension of **t**. We can use this function to get the list of time when the *i*th component of **ALL** (*t*) has the value *y*.

ALL (*t*) is stored in the form of **t** = g(i, y), since it will be used most frequently in our algorithm. g(i, y) can also be rewritten as below, where ToVector({Xi}) := [X1,…,Xn].

$g(i, y) = \text{ToVector}(\{t | \forall k \le \text{num}(t = q_k^{-1}(y) \text{ iff } (q_k^{-1}(0) - y) \bullet (y - q_k^{-1}(T_{k+1}) \ge 0))\})$

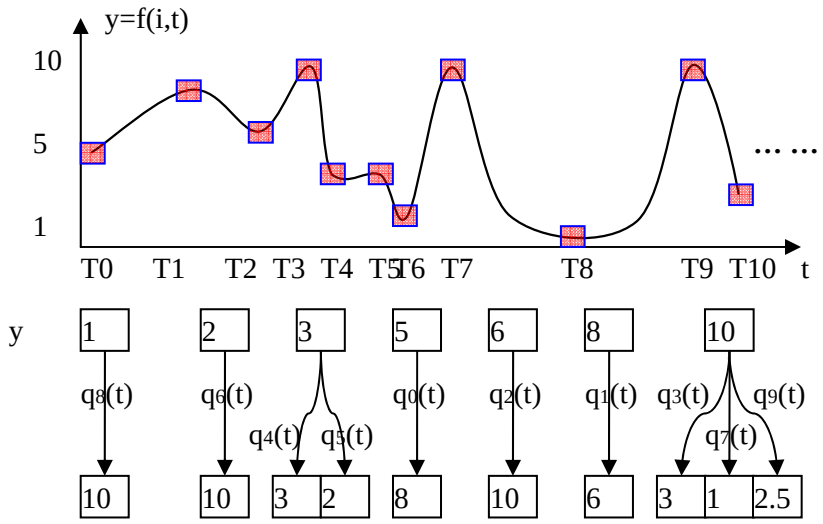

Figure 8.4 Data presentation of g (i,y)

Figure 8.4 shows the data presentation of g (i,y) (*i* is some fixed value.). E.g. in the above curve, g(i,10)=[3,7,9]. When implementing this data presentation, each $q_i(t)$ can be further compressed using parameterized analytical equations, when a certain tolerance of error is allowed. In other words, some sub curve $q_i(t)$ can be combined, if they are similar. An integer counter C_i is associated with each curve $q_i(t)$, which will be increased if our algorithm hits on the sub curve. Some rules are used to delete the sub curves from g(i,y) whose counter value is comparatively small in order to keep the size of g(i,y) constant when the length of curve increases ($t \to +\infty$).

Now we are ready to give the ST rules which are expressed in the form:

att (t+Δt) = ST3(ST2 (ST1(**ALL**(t))), **att**(t)), where ST1, ST2, ST3 are three rules used to modify the **attention** for the motion generation system. Each rule is given below with their explanations.

ST rule 1

$$ST1(f_i(t)) := \sum_{k=0}^{maxlen} \left(\frac{1}{(k+1)\sqrt{2\pi}\sigma} e^{-(t-g(i,f_i(T_c+(1-k)\Delta T)))^2/(2\sigma^2)}\right)$$

ST1 searches each f(i,t) of **ALL**(*t*) for the longest sequence of matching sub curves in the form $f_i(t)$, $t \in [T_c+\Delta T-\Delta Len, T_c+\Delta T]$,($\Delta Len=\Delta T \times maxlen$). ST1 returns a value for each *t* ($t < T_c+\Delta T-\Delta Len$), indicating how the curves near *t* resembles the latest occurring curves. The colored boxes in Figure 8.4 show the effect of ST1. The curves in these boxes resemble the latest occurring curves (its color also shows the degree of similarity). Figure 8.5shows the curve of ST1 for a sample f(t).

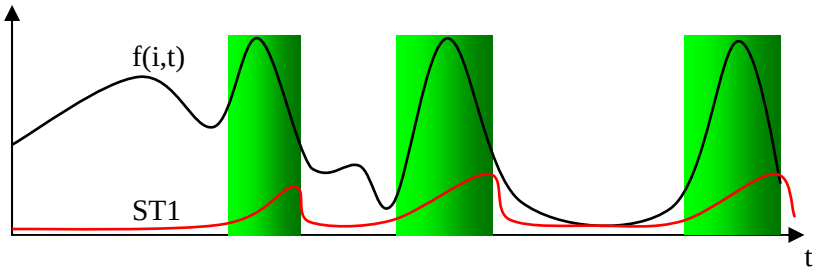

Figure 8.5 Sample curve of ST1

ST rule 2

$ST2(\mathbf{x}(t)) = \sum_{i=0}^{N-1} x_i(t)$, where **x** is an *N* dimensional vector of functions

ST2 is the rule used to find synchronizing positions in a set of functions of *t*. In the algorithm, the result from ST1 is used as input of ST2, i.e. ST2 (ST1 (**ALL** (t))). It does so by simply summarizing the components of **x**(t). The result of ST2 is just a single function of time, showing the places in the time domain where the reappearing patterns of **x**(t) occurs. Figure 8.6 shows the sample curve of ST2. The apex of its convexes denotes the center of synchronization position. And its magnitude roughly denotes the number of synchronization component. In ST theory, maximums in the curve of ST2 are the places in the time domain, where different kinds of simulations may occur. The **attention** mechanism will select among these simulations to decide the next output for the motion generation system.

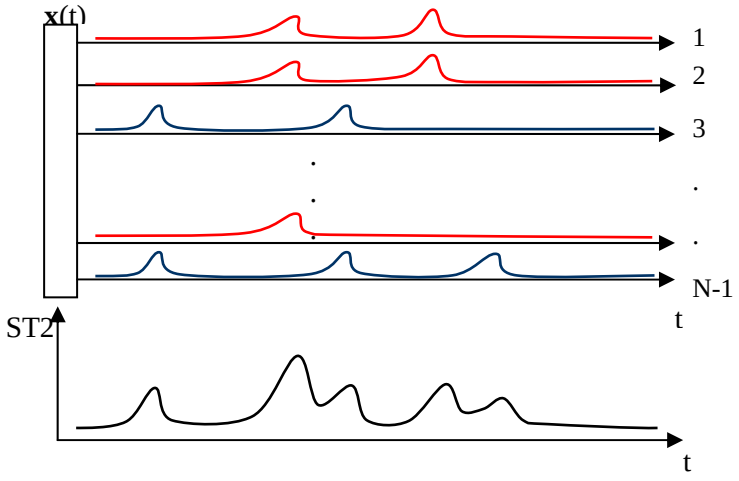

Figure 8.6 Sample curve of ST2

ST rule 3

$ST3(x(t), att(t)) = \text{attention}(\text{divide}([\text{sort}\{T_i\}\text{by}\int_{T_i}^{T_i+\Delta T} x(t)dt], att(t)))$, where {Ti} are the maximums in the curve of x(t). (x(t) is usually the result of ST2.)

ST3 is the rule used to generation the **attention**. It first sorts {Ti} by the area of a small region after Ti along the x(t) curve. Then it use a *divide* function to partition all f(i,t)

into $\{\{f_{a_{i,j}}(t)\}_j\}_i$ according to **att**(t) and the sorted maximum points. In Figure 8.6, a possible division would be {{f(1,t), f(2,t), f(N-1,t)}, {f(3,t), f(N,t)},…}. Each group is synchronized by one of the time in {Ti}. After deciding this division, the new **att**(t) are generated, so that the attention for components in the *i*th group is raised to a value at least higher than all other positions in the time domain of that component. Components in the same group form a specific *simulation* which can be used to generate subsequent motions for these components. A general degradation is performed in **att**(t), so that the total amount of attention is a constant. The division function is affected by both the result from ST2 and the last distribution of attentions. Because the *divide* function is flexible, different motion generation systems might adopt different *divide* criterion and no details of it will be given here. However, as a general rule, we should try to minimize the total number of groups (simulations) during the division. The **relation** matrix can be used in the *divide* process to minimize the number of groups. E.g. components of high relevancy tend to be included in the same group.

8.4.3 A Discussion of Performance

When implementing the algorithm, a lot of things can be simplified. The data structures for ST1, ST2 and ST3 can be in the form of list of key points, since the information in their curves are few compared to the length of their time domains.

The space complexity of the algorithm is O(*Fea*N*). *Fea* is the average number of reoccurring patterns (simulations). *N* is the number of dimensions in **ALL**(*t*).

The time complexity of ST1, ST2, ST3 can be O(log(*Fea*)* *Fea**N), O(N), O(N*N*N) respectively. So the total time complexity of the algorithm is $O(\mathrm{Fea}\times\log(\mathrm{Fea})\times N+N^3)$. In a typical case of humanoid animation system, *Fea* is usually several hundreds and *N* is usually round 50. It is possible to use the motion generation algorithm for real-time character animation.

8.4.4 Implementation

This section is not implemented. I will give the implementation suggestion from its realization in a simple computer game engine we developed.

8.4.5 Application in the Game Engine

In this section, I will discuss how the research fits into the large context of automatic motion synthesis in the computer game engine which includes both global and local animations.

8.4.6 Introduction to Automatic Motion Synthesis

First of all, the proposed motion generation algorithm is not the replacement to all local animation solutions in a computer game engine, due to performance and other requirements such as precision and ease of development. It is up to the game developers to decide when and which characters are going to use a certain kind of animation engine.

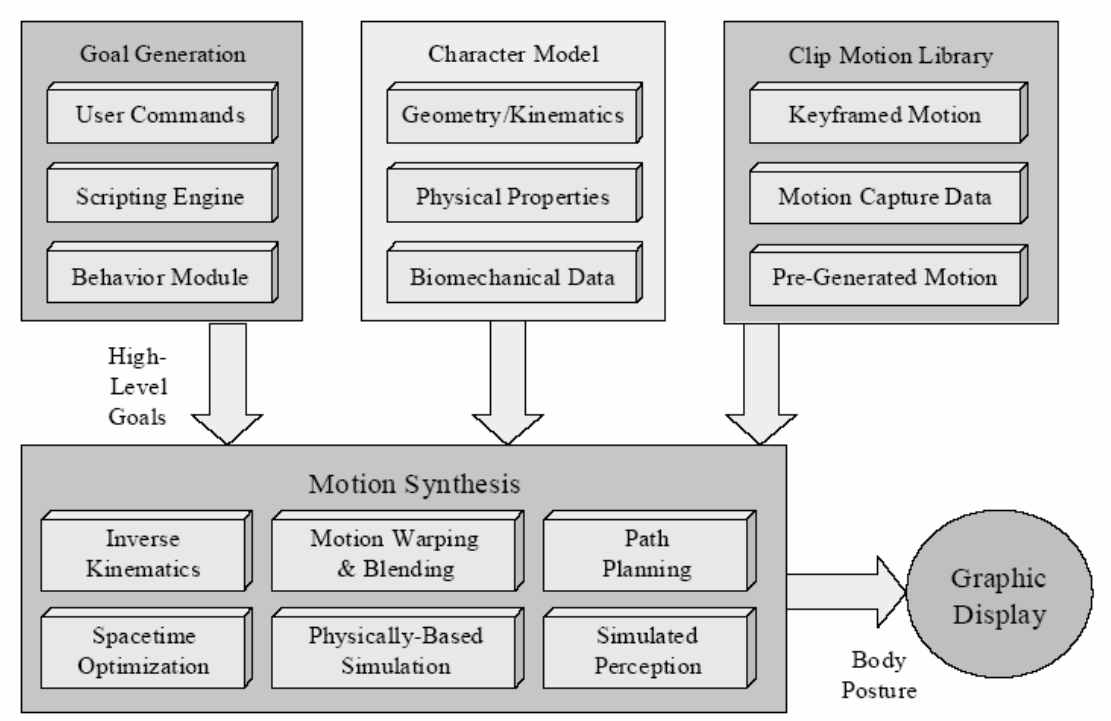

Figure 8.7 Resources available for motion synthesis
Figure taken from [6]

Generally speaking, the new motion generation system is used in the game engine only for the main character(s) when it is in the view frustum. Now, we are going to see how to divide the control variables relevant to the character between the global motion planning and local motion animation.

Figure 8.8 (left) shows a typical character. For fast motion synthesis, we want to only consider planning for the degrees of freedom that affect the overall global motion of the links of a character (i.e. the root joint DOFs). For the joint hierarchies we used, each of the 6 degrees of freedom of the Hip joint falls into this category. The idea is to compute motion plans for these DOFs (which we shall refer to as active DOFs or active joints), while the remaining joints (which we refer to as passive DOFs or passive joints) are either held fixed or controlled by a local animation algorithm. For our navigation strategy, passive joints are animated either by the new motion generation system or by cycling through a clip motion locomotion gait. However, passive joints could also potentially be driven by the active joints with their motion computed using a simple mathematical relation or even a sophisticated physically-based simulation (e.g. the motion of a character's hair in response to the gross motions of the head. In these cases, I call them add-on animations.).

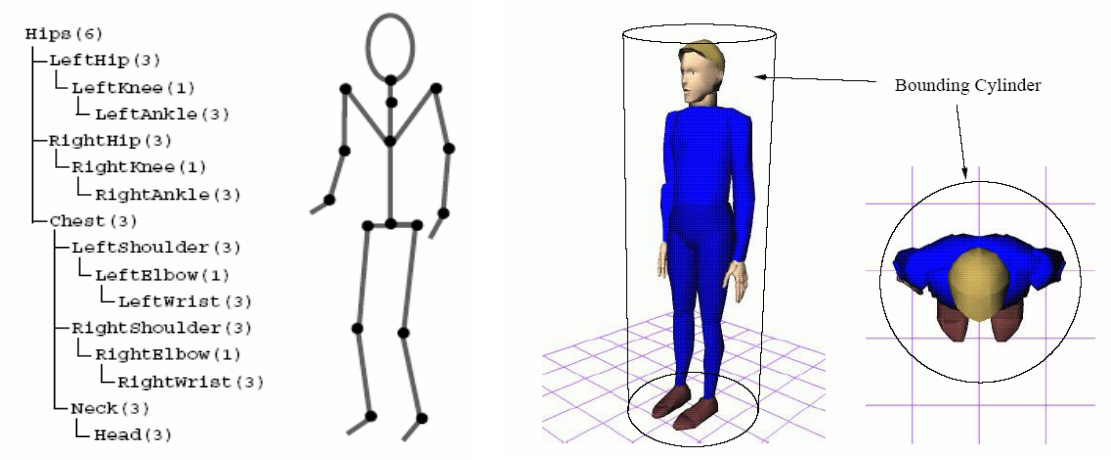

Figure 8.8 The major joints in the kinematic hierarchy used for locomotion are shown, along with the number of DOF for each joint.
Figure taken from [6]

Moreover, we can further reduce the dimensionality (and hence the efficiency) of the planning phase by considering only a subset of the active joints when possible. For instance, if we assume that the character navigates on a surface, we can omit one of the active translational DOFs (the height of the Hip joint above the surface), as well as two of the active rotational DOFs (by assuming the figure's main axis remains aligned with the normal to the surface). Suppose all of the passive DOFs and the omitted active DOFs are driven by a controller that plays back a simple locomotion gait derived from motion capture data. By approximating the character by a suitable bounding volume (such as a cylinder), that bounds the extremes of the character's motion during a single cycle of the locomotion gait, we have effectively reduced the navigation planning problem to the 3-dimensional problem of planning the motion for an oriented disc moving on a plane. Figure 8.8 (right) shows one of the characters used in our experiments along with its bounding cylinder.

In case we do not want to generate location from recorded motion clips, we can use the new motion generation algorithm. The control variables that should be taken from the global animation are the velocities for all the DOFs of the character’s root joint (Hips). Behavior commands can also form a special dimension in the algorithm. The environmental variables such as a sound source or terrain features relative to the character or its emotions can also be included as different dimensions in the algorithm. All these additional dimensions are overridden by the **User** (as is called in our algorithm) and used to for the motion generation system to produce motions for the rest of the dimensions.

8.5 Conclusion

AI in computer game engine can be simple, orthodoxy and very sophisticated. I believe computer game engine is an ideal platform for researches in artificial intelligence.

Chapter 9
Frame Rate Control

9.1 Introduction

Several new challenges arise in the visualization and simulation of distributed virtual environment of unsteady complexity, such as in a computer game engine. 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.

Timing or frame rate in game engines is both unpredictable and intertwined. For example: some scene entities are animated independently, whereas others may form master-slave animation relations; 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 is sometimes less satisfactory than time-scaled animation, which will be discussed in the paper. 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 processes and game objects. In other words, time management architecture (such as the one proposed in this paper) should be carefully 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.

Section 2 discusses time related issues in game engine as well as related works. Section 3 formulates the time management problem and proposes the frame rate control architecture. Section 4 evaluates the system by examples in our own game engine. Section 5 concludes the paper.

9.2 Timing in Game Engine and Related Works

Game related technologies have recently drawn increasing academic attention to it, not only because it is highly demanding by quick industrial forces, but also because it offers mature platforms for a wide range of researches in computer science.

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. To our knowledge, there have been no open-source game engines which directly support time management so far. 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 phenomenons are caused by a number of coordinating modules in the game engine and cannot be easily solved by a simple modification.

This section reviews the current implementations of a number of time-related modules in game engines. We have discussed them in a way that practitioners can figure out how to address them with the proposed frame rate control architecture, which is presented in Section 3.

9.2.1 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). Bryson’s paper [52] addresses time management issues in these environments.

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, stop or reverse the apparent flow of time (without slowing or stopping the graphics process), while still allowing direct manipulation exploration within an individual time step.

However, decoupling graphics and computation is a way to explore parallelism in computing resources (e.g. CPU, GPU), 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.

9.2.2 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 [52] 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).

9.2.3 Frame Rate and LOD

The largest number of related work [53] [54] lies in Frame rate and LOD. However, the proposed framework does not directly deal with how the LOD optimization module should react to feedbacks from the current frame rate; instead it aims to provide a preferred activation rate to modules not limited to LOD optimization.

9.2.3.1 Frame Rate and Scene Complexity

In many situations, a frame rate, lower than 30 fps, is also acceptable by users as long as it is constant. 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 contained in BSP nodes), 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.

9.2.4 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 Cronin's paper [55], a number of time synchronization mechanism for distributed game server are presented, with its own trailing state synchronization method. Diot [56] presented a simple and useful time synchronization mechanism for distributed game servers.

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 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.

9.2.5 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 article [11] provides a comprehensive overview about the time related problems with the use of a physics engine in game development.

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 in Section 4 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.

9.2.5.1 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. With time management, we can use two synchronized frame rate controllers to reduce the amount of computations. One controller is assigned a low frame rate for updating the physical parameters of an animated object; the other is assigned the same (high) frame rate as the simulation time to interpolate the object's physical parameters and feed to the physics engine.

9.2.6 Conclusion to Related Works

Section 2 discussed a number of places in the game engine where time management is critical, as well as related works on them. Time management should be carefully dealt with in a computer game engine. In fact, we believe it will soon become common in the backbone system of distributed computer game engines.

9.3 Frame Rate Control Architecture

In this section, we propose the Frame Rate Control (FRC) architecture and show how it can be integrated in an actual computer game engine.

9.3.1 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) → {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 $\{t^{s_k} \mid k \in N, \lim_{\delta x \to 0}(\frac{f(t^k) - f(t^k - \delta x)}{\delta x}) = +\infty\}$ be a set of points on *t*, where the value of f(t) changes from 0 to 1. Let $\{t^{e_k} \mid k \in N, \lim_{\delta x \to 0}(\frac{f(t^k + \delta x) - f(t^k)}{\delta x}) = -\infty\}$ 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 9.1 shows the curves of three related frame rate functions: i, j and k.

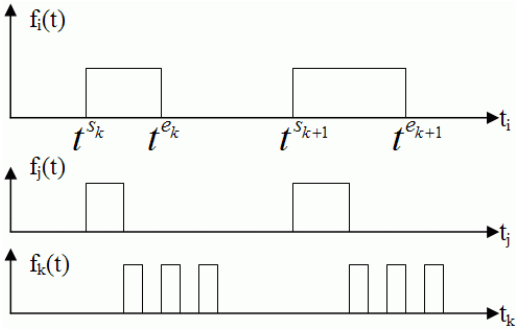

Figure 9.1 Sample curves of frame rate functions

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. $\forall_{k,l}((t_i^{s_k}, t_i^{s_{k+1}}) \bigcap (t_j^{s_l}, t_j^{s_{l+1}}) = \varnothing)$. 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/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. $\forall k((t^{s_k} - t^{s_{k-1}}) = ConstIdealStep)$. 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. $\forall k((t^{s_k} - t^{s_{k-1}}) <= ConstIdealStep)$. 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.

9.3.2 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 constraints listed in Section 3.1. Instances of FRC Controller are created and managed in a global place (such as in a singleton class for time management). A set of global functions (see Time Scheme Manager in Figure 9.2) 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 (see Table 3.1) 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. Moreover, most present day games are composed by commercial engines whose base programming interface is fixed or unadvised for modification. 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 9.2), 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.

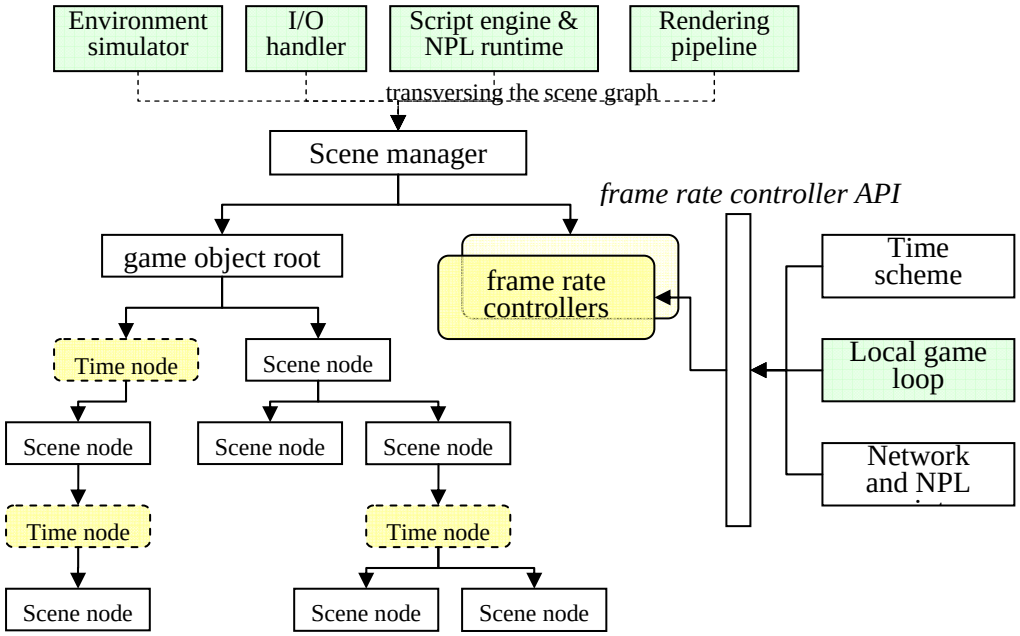

Figure 9.2 Integrating time control to the game engine

Like most computer game engines, we use a scene manager for efficient game objects management. Our goal is to associate appropriate FRC controllers to each scene objects. However, keeping a handle or reference in each scene object is inefficient in terms of management and memory usage. Moreover, most present day games are composed by commercial engines whose base programming interface is fixed or unadvised for modification. Fortunately, we can achieve our goal by taking advantage of the tree-like hierarchy in the scene manager and its transversal routines during rendering and simulation. This is done by creating a new kind of dummy scene node called *time* node, which contains the reference or handle to one or several FRC controllers. Then, we insert these time nodes to the scene graph as any other scene nodes. Finally, we use the following rules to retrieve the appropriate FRC controllers for each scene object:

- The FRC controllers inside *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, we adopt the settings in FRC controllers which are nearest to the current node.

Since frame rate controllers are managed as top-layer (global) objects in the engine (see Figure 9.2). Any changes made to the FRC controllers will be immediately reflected in the next scene traversal cycle. Table 3.1 shows the game loop. 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.

9.4 Evaluation

This section contains some use cases of the proposed framework in ParaEngine. 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.

9.4.1 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 9.1. 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 9.1 Frame rate control schemes

	N-scheme	**C-scheme**
ConstIdealFPS	**30 or 60**	**20 or 25**
Rendering	FRC_CONSTANT	FRC_CONSTANT
I/O	FRC_CONSTANT	FRC_CONSTANT
Sim & scripting	FRC_FIRSTORDER	FRC_CONSTANT

9.4.2 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.

9.5 Conclusion

Time management is very important in the visualization and simulation of distributed virtual environment, such as networked computer game worlds. The paper reviews a number of time-related issues in computer game engines and proposes a unified frame rate control architecture which can be easily applied to a computer game engine. The frame rate system has been successfully used in our own distributed game engine and may also find applications in other multimedia simulation systems.

Chapter 10
Conclusion

10.1 A Discussion and Evaluation

In this literature, I introduced the prospect of distributed Internet games, proposed the simulation framework, discussed a sample computer game engine (ParaEngine) and explained using NPL scripting language to compose distributed game world. The idea came from the observation that our brain is both a distributed computing environment and a theatre of multimedia (internal perceptions). The analogy of human cognition to simulation system has been applied to a computer game engine to construct distributed Internet games. Bringing networked virtual game worlds and game world logic to the open Internet will spawn new types of computer games.

Distributed game engine is an open game engine framework for distributed and dynamic game content and logics on the Internet. Current main stream Internet is based on text and web page browsers. Yet, the Internet has been a highly successful invention to join these distributed web content with hyperlinks. It is intuitive to hope for the Internet to transform from 2D presentation to a mixture of 2D and 3D. I believe the driving force that is going to push this transformation is Internet 3D computer games. Computer games can also bring about a series of civil and commercial 3D virtual reality services to our everyday life.

Since February 2004, the concept of distributed game engine has been proposed. Since March 2004, I have researched and implemented the first distribute game engine called ParaEngine. From technological viewpoints, ParaEngine differs from traditional game engines in the following aspects: (1) all game world content and logics can be created and controlled through scripts, (2) it supports network transparent scripting system and (3) it supports unbounded, dynamic and extensible 3D game worlds as well as physics and AI simulation spanning the virtual worlds.

10.2 The Parallel World Game

Parallel World is a distributed 3D 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. NPL is the enabling technology to script the ever evolving game world logic on the network. Parallel World project is initiated by me and it is currently being developed by a group of students at Zhejiang University. Since game development involves tremendous works on art, game rules, and level design, my task is focused on the game engine and technical support regarding the integration process. For example, after the game plot has been discussed and written, the level designers lay out the background of the story; then the 2D artists draw sketches of the game world on pieces of paper; then the 3D artists model the game world in 3DSMax, meantime the animation artists make skeletons for the game characters and hand-animate them; finally the level designers use these game assets to compose the game world through the scripting system of the game engine. Given a collection of art resources, everyone is capable to extend the game world using the script system; a newly composed game world may include logics from the network and be part of the virtual world on the Internet.

I welcome everyone to download our tech demo from my personal website:

http://www.lixizhi.net/paraworld/web/index.htm

Figure 10.1 shows some old and recent screen shots of the ParaEngine technology demo. Since we have not started the game development formally, there are no screen shots for the Parallel World game.

Figure 10.1 ParaEngine Tech Demo Snapshot

10.3 Future Works

When I was writing this literature, ParaEngine was still under active development. And I have planned to develop a convincing and playable game to demonstrate the potential of distributed computer game engine.

Currently, ParaEngine is not as complete as a commercial game engine. For example, its peripheral composing tools (such as 3D model and animation exporters, map and terrain exporters, script debugger, etc) are not fully developed. It does not support many new features from current graphic acceleration cards. But I believe, in the process of building the first distributed game demo, our team could solve these left over problems. In the middle of 2006 (a year from now), I plan to release our first game demo based on ParaEngine. And of course, it will be a free game for everyone.

10.4 Immersive Games: A Scheduled Dream

It is no longer fiction now to be immersed in the virtual reality world. In five to ten years time, we can hope to experience it everyday in our life. Since young, I have seen quite a few research projects on its hardware facilities completed in my father's institute, and I myself am doing researches on its software frameworks in recent years. From my observation, there is no technological bottle neck for the virtual reality dream to come true. Lots of inventions in yesterday' dreams are on today's schedule. I used to ask myself: is there a shortest path to reach the dream? My recent answer shall be *by creating immersive games*.

Appendix A:
ParaEngine Feature Lists

Feature lists:

Supported OS:	Windows
API:	DirectX 9.0c
Language:	C/C++ (VS.NET 7.1)
Features:	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.
- The new scripting paradigm implies new ways of implementations on the following aspect of game development: Internet game and game society establishment and maintenance, non-linear stand-alone AI system and networked AI systems, 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.

Graphic engine, AI and physics:

- ParaX file format: Skeletal model animation control and rendering
- Garbage collected 2D GUI engine, fully compatible with the scripting engine.
- Large scene management and rendering system. It is most suitable for games that call for the following features at the same time: infinitely large map of any shape, many inter-related movable characters acting in various locations of the map, complex map script triggering and story design.
- Optional ROAM algorithm terrain engine. In-game map/terrain creation and modification; auto-breaking large texture files, multi-texturing on terrain surface; fast ray-tracing; automatically snapping objects to terrain surface, automatic latticed based terrain, raw elevation file.
- Sky and Fog.
- Robust shadow casting.
- Game resource management system: graphics (bmp, tga, jpg, blp) 3D objects (x file, mdx file), sprite animation (bmp, tga, jpg), 3D skeletal animation (ParaX animated file, X file format), sound(wav, mp3), multi-file management (mpq)
- Real-number coordinate system based physics: path-finding, collision detection and response. Currently it supports spherical objects and rectangular object of any rotation. Integration with the Novodex physics engine.
- Three customizable automatic follow cameras and one 360 degree free camera. Occlusion constraint is applied to all camera modes.
- Mouse-ray picking for all scene objects.
- Video and still image capturing.
- Frame rate control.

Examples:

“Parallel World“: 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 severs. ParaEngine is a simple yet comprehensive 3D computer game engine developed for this game.

Summary: ParaEngine is a simple yet comprehensive 3D computer game engine. It is 100% new design and coding.

Developed by Xizhi Li: LiXizhi@zju.edu.cn or lxz1982@hotmail.com

Appendix B:
A Startup Tutorial of NPL Scripting Language

This tutorial is taken from [self: 26]. For more NPL tutorials, please refer to that document.

Author(s)	Date	NPL Version	Logs
LiXizhi	2005-4-20	0.5.8 or later	

Description	You will learn the fundamentals of using NPL scripting language to construct a local game world on your personal computer.
Skill Level	Rudimentary
Time	40-60 minutes
Functions	- NPL scripting concepts - A sample script file - Editing environment - The main game loop - World creation API - A “hello world” script demo
Files	[1] \Script\Tutorials\HelloWorld.lua [2] \Script\gameinterface.lua [3] NPL scripting reference manual (PDF or HTML version) [4] _thirdparty tools\ LuaEdit\Bin\LuaEdit.exe [5] _thirdparty tools\ LuaEdit\Bin\Help\ Lua Manual.pdf

NPL scripting concepts

In distributed game engine, it is common to deal with thousands of interactive entities on the local runtime and among its web servers. Our solution is based on scripting technology, which has already been broadly used in modern computer game engine to provide flexible and extensible integration of a single game engine runtime and the rest of the world.

Scripting system is a high-level language and runtime environment which sits between the core of a game engine and the human beings as well as intelligent agents. Briefly speaking, the following workflow applies in some game engine configurations. For each game engine instance on the network, it will (1) update changed game states from input sources, (2) simulate game world objects which contain local physical properties, (3) generate physics events and other feedbacks to intelligent entities for decision making, and (4) render the portion of the world within the camera view to the display. In (1), a large portion of game engine input (with the exemption of direct scene manipulation) comes from the scripting system. In (3), most events and feedbacks (perhaps only with the exemption of local haptic devices) are reported to the scripting system runtime.

The scripting system consists of a collection of activation files as well as other reference files. An activation file may be activated by other script files, some synchronization timers and the game engines. Logics are usually scripted (written) in activation files and carried out

by series of file activations, during which process activated files may emit UI commands to game engines for game state updates.

A sample script file

A script (or activation) file should at least contain a function body called *activate* (code). This is like the main () function in C programs. Once a script is activated either by other scripts or by the local game engine, the *activate* function will be called by the script runtime. All script language must also support an asynchronous function called *NPL.activate* (sScriptPath, sCode) which takes two explicit parameters. The first parameter is a path name of the script file which will be activated; the second parameter is a program code to be executed in the global space of the specified script file. The sCode parameter may be used to transmit data to another script. The function will return immediately, and its specified script will be activated in the next time step or if the specified script is on a remote computer, the script runtime will automatically send it via available connection or establish a new one. A sample script code is given below.

File name: SampleScript.lua

```
function activate(code)
  if (state == nil) then
    … …
    local player = ParaScene.CreateCharacter ("Actor", "biped mesh", "", true, 0.5, 0, 1.0);
    … …
    ParaCamera.FollowObject(player);
    -- activate another script.
    NPL.activate("ABC: \\pol_intro.npl", "state=nil;");
  else
    … …
  end
end
function Service1()
  … …
end
global items[]; -- a global variable (table)
```

In the above code, Para*() functions are game specific UI functions (or visualization commands). These UI functions cover a complete set of game content controllers, such as loading game scene objects, displaying dialog boxes, playing sounds, updating player positions, animating characters, changing camera mode, etc. In the *NPL.activate* function, the file path parameter contains the string “ABC:” which is a namespace shortcut that will be

converted to an actual network address by the script runtime during execution. If the short cut is omitted, the script engine will regard it as local and search for it in the local directory. A script file is also allowed to contain any number of global functions called services, such as Service1 () in the sample. Services and all global variables are accessible by other script files via the sCode parameter of the *NPL.activate* function, whereas everything inside the *activate* function are private to the script file.

Editing environment

There are several third party tools for editing scripts, such as LuaQDE and LuaEdit (they can be found in the third party tools directory). Normally, a traditional text editor will do. However, with a script IDE, one can compile the code for syntax errors before running it in the game engine. Currently, ParaEngine will neither report nor handle any errors in script files due to performance reasons of the game engine; hence a simple syntax error in script might cause the game engine to break down unexpectedly. And this is why we recommend an IDE for beginners.

In additional to this tutorial document, script writers may constantly refer to the NPL script reference manual for specific instructions of world constructing functions. The NPL reference manual is also included with the game engine. It is available in both PDF and HTML format. NPL is an extended language over Lua, which provides a light-weighted general programming interface and fast runtime environment. Please refer to Lua manual (also included in our tools directories) for information about this language system.

Once scripts are written and successfully compiled, it can be tested in the game engine. Here *compile* is for error checking only; the game engine will dynamically compile script code at real time, when they are loaded.

The main game loop

Each game engine instance will call a default script named “gameinterface.lua” several times per second. This script is the default game loop script and the entry point of the scripting system. Other scripts may be loaded from it. At the end of this tutorial, we will run a simple script called “HelloWorld.lua” from it.

World creation API

The ParaEngine World Creation APIs are organized into the following namespaces:

NPL	Neural Parallel Language functions are in this namespace
ParaGlobal	a list of HAPI functions to globally control the engine
ParaAsset	a list of HAPI functions to manage resources(asset) used in game world composing, such as 3d models, textures, animations, sound, etc
ParaScene	a list of HAPI functions to create and modify scene objects

	in ParaEngine
ParaCamera	The camera controller

The following code will load an animation file from disk and create a character from it at a specified location in the game world. Please note that functions in a namespace can be accessed using the syntax: *{namespace name}.{Function name}*.

Sample code snippets

```
--create asset or resources used in this scene
ParaAsset.LoadMultiAnimation("HeroAsset", "xmodels\\moon_female.x");
ParaAsset.Init();


-- create global character. It can walk in the entire scene
local player = ParaScene.CreateCharacter ("girlPC", "HeroAsset", "", true, 0.35, 3.14159, 1.5);
player:SetPosition(45, 0, 45);
player:SnapToTerrainSurface(1);
ParaScene.Attach(player);
```

A “hello world” script demo

We will create a simple scene with a user-controllable character in the scene and display a text called “Hello world!!!” to the display when the scene is loaded. In order to load the script we add the bold line in the following code snippet in to the “script\gameinterface.lua”. This will cause our HelloWorld.lua script to be loaded when the game is started.

File name: script\gameinterface.lua

```
function activate(intensity)
        if(state==nil) then
                … …
                dofile(“\script\tutorials\HelloWorld.lua”);
        elseif(state==0) then
        … …
end
```

Then we will create a file called HelloWorld.lua in the script\tutorial directory as below.

File name: \script\tutorials\HelloWorld.lua

```
--create asset or resources used in this scene
```

```
ParaAsset.LoadMultiAnimation("HeroAsset", "xmodels\\moon_female.x");
ParaAsset.Init();


-- create global character. It can walk in the entire scene
local player = ParaScene.CreateCharacter ("girlPC", "HeroAsset", "", true, 0.35, 3.14159, 1.5);
player:SetPosition(51, 0, 51);
player:SnapToTerrainSurface(1);
ParaScene.Attach(player);


-- set the camera to follow him
ParaCamera.FollowObject(player);
ParaCamera.ThirdPerson(0, 2,0,0.7);


-- Display the “hello world” text
local MyWindow = ParaUI.CreateWin("HelloWorldWindow", "_lt", 20,100, 600,420);
MyWindow:SetText([[  Hello World !!!  ]], 1677215, "");
MyWindow:AttachToRoot();
```

Now, we are ready to start the game engine and watch our character alive.

Bibliography

[1] S Singhal and M Zyda, *Networked Virtual Environments: Design and Implementation*, ACM Press, 1999.

[2] *Web3D Consortium*, http://www.web3d.org/.

[3] Manninen T, *Interaction in Networked Virtual Environments as Communicative Action - Social Theory and Multi-player Games*, *CRIWG*, IEEE, 2000.

[4] Axel Buendia and Jean-Claude Heudin, *Towards Digital Creatures in Real-Time 3D Games*, Virtual Worlds (2000), pp. 44-53.

[5] Marc Evers and Anton Nijholt, *Jacob - An Animated Instruction Agent in Virtual Reality*, *ICMI*, LNCS 1948, 2000, pp. 526-533.

[6] J. J. Kuffner, *Autonomous Agents for Real-Time Animation*, Stanford, 1999.

[7] Pedro Morillo, *A grid representation for Distributed Virtual Environments*, *European Across Grids Conference 2003*, 2003, pp. 182-189.

[8] Marko Meister and Charles A. Wuthrich, *On Synchronized Simulation in a Distributed Virtual Environment, In Proceedings WSCG 2001*, 2001.

[9] *Novodex physics engine*, http://www.ageia.com/novodex.html.

[10] *Havok solutions*, http://www.havok.com.

[11] *Outsourcing Reality: Integrating a Commercial Physics Engine*, *Game Developer Magazine*, 2002.

[12] *OGRE Project Site*, http://www.ogre3d.org.

[13] *3D Engines Database*, http://www.devmaster.net/engines/.

[14] Xizhi Li and Qinming He, *WAF: an Interface Web Agent Framework*, *International Conference on Information Technology*, 2004.

[15] Xizhi Li, *DHCI: an HCI Framework in Distributed Environment*, *11th International Conference on Human-Computer Interaction*, 2005.

[16] Xizhi Li, *Using Neural Parallel Language in Distributed Game World Composing*, *Distributed Framework of Multimedia Applications*, IEEE conf, 2005.

[17] Emmanuel Frécon and Mårten Stenius, *DIVE: A scaleable network architecture for distributed virtual environments*, Distributed Systems Engineering Journal (DSEJ), vol. 5 (1998), pp. 91-100.

[18] S Benford, C Greenhalgh and D Lloyd, *Crowded Collaborative Virtual Environments*, ACM CHI'97 (1997).

[19] Jim Brodie Brazell, *Digital Games: A Technology Forecast*, Texas State Technical College, 2004.

[20] G Hesslow, *Conscious thought as simulation of behaviour and perception*, Trends in Cognitive Sciences, vol.6 (2002), pp. 242-247.

[21] Murray Shanahan, *The Imaginative Mind A Precis*, *Conference on Grand Challenges for Computing Research*, 2004.

[22] Robert Dunlop, *Writing the Game Loop*, Microsoft DirectX MVP site: http://www.mvps.org/.

[23] Gerhard Reitmayr, *Flexible Parametrization of Scene Graphs*, Virtual Reality (2005).

[24] Lars Bishop, *Designing a PC Game Engine*, IEEE Computer Graphics and Applications, archive vol.18 (1998), pp. 46-53.

[25] Lasse Staff Jensen and Robert Golias, *Deep-Water Animation and Rendering*, http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm, 2001.

[26] Vladimir Belyaev, *Real-time simulation of water surface*, *GraphiCon-2003 Conference Proceedings*, MAX. Press, 2003, pp. 131-138.

[27] Microsoft Corp, *Direct X file Format Specification*, Microsoft, 2002.

[28] Frank Luna, *Skinned Mesh Character Animation with Direct3D 9.0c*, 2004.

[29] *Deep exploration v3.5*, Right Hemisphere, 2004.

[30] Don Burns, *OpenSceneGraph*, http://openscenegraph.sourceforge.net/index.html, 2005.

[31] D Fritsch, *3D Building Visualization - Outdoor and Indoor Applications*, Photogram metric Week'03 (2003), pp. 281-290.
[32] P Lindstrom, *Real-Time, Continuous Level of Detail Rendering of Height Fields*, *Proceedings of SIGGRAPH'96*, 1996, pp. 281-290.
[33] *An extensive overview of terrain rendering*, http://www.vterrain.org/.
[34] Tatcher Ulrich, *Super-size it! Scaling up to Massive Virtual Worlds*, (course notes) SIGGRAPH (2002).
[35] J CARMACK, *E-mail to private list*, Published on the NVIDIA website, 2000.
[36] Samuel Hornus, *ZP+: Correct Z-pass Stencil Shadows*, *ACM Symposium on Interactive 3D Graphics and Games*, April 2005.
[37] Eric Chan and Fredo Durand, *Rendering Fake Soft Shadows with Smoothies*, *14 th Eurographics Symposium on Rendering*, 2003.
[38] Andrew Woo, Pierre Poulin and Alain Fournier, *A survey of shadow algorithms*, IEEE Computer Graphics and Applications, vol 10 (November 1990), pp. 13-32.
[39] Morgan McGuire, *Fast, Practical and Robust Shadows*, (2004).
[40] Frank Crow, *Shadows Algorithms for Computers Graphics. Computer Graphics*, Proceedings of SIGGRAPH 1977, Vol. 11 (July 1977).
[41] Cass Everitt and Mark J. Kilgard, *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, developer.nvidia.com, 2002.
[42] NovodeX AG, *NovodeX SDK Integration Guide*, www.novodex.com, 2005.
[43] WH Bares, S. Thainimit and S. McDermott, *A Model. for Constraint-Based Camera Planning*, *Proc. of the. 2000 AAAI Symp*, 2000, pp. 84-91.
[44] Nicolas Halper, Ralf Helbing and Thomas Strothotte, *A camera engine for computer. games: Managing the trade-off between constraint satisfaction and frame coherence*, *In Computer Graphics Forum: Proceedings EUROGRAPHICS*, 2001, pp. 174-183.
[45] Steven Mark Drucker, *Intelligent Camera Control for Graphical Environments*, MIT, 1994.
[46] Mira Dontcheva, *Layered Acting for Character Animation*, *The proceedings of ACM SIGGRAPH*, 2003.
[47] R. Ierusalimschy, *Alua: An event driven communication mechanisms for parallel and distributed programming*, PDCS-99 (1996), pp. 108-113.
[48] M. Girard and A Maciejewski, *Computational modeling for the computer animation of legged figures*, *Proc. of SIGGRAPH '85*, 1985.
[49] R. Boulic, D. Thalmann and N. Magnenat-Thalmann, *A global human walking model with real time kinematic personification*, The Visual Computer, vol 6(6) (1990).
[50] Ying Liu, *Interactive Reach Planning for Animated Characters using Hardware Acceleration*, U. Penn, 2003.
[51] J. C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers.Boston, MA, 1991.
[52] S. Bryson and SandyJohan, *Time management, simultaneity and time critical computation in interactive unsteady visualization environments*, IEEE Visualization '96 (1996).
[53] Thomas A. Funkhouser and Carlo H. Séquin, *Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments*, *Computer Graphics (SIGGRAPH '93 Proceedings)*, August 1993, pp. 247--254.
[54] Markus Grabner, *Smooth High-quality Interactive Visualization*, *Proceeding of SCCG 2001*, April 2001, pp. 139-148.
[55] E. Cronin, B. Filstrup and A. R. Kurc, *A distributed multiplayer game server system*, UMich: http://www.eecs.umich.edu/~bfilstru/quakefinal.pdf, 2001.
[56] C. Diot and L. Gautier, *A Distributed Architecture for MultiPlayer Interactive Applications on the Internet*, *In IEEE Network magazine*, August 1999.

ParaEngine Documentation References

The following is ParaEngine document written by me:

[1] ParaEngine Outline.doc

[2] ParaEngine and GUI & OPC.doc

[3] ParaEngine and MeshAnimationFile.doc

[4] ParaEngine and SpecialEffects.doc

[5] ParaEngine and new pipeline.doc

[6] ParaEngine and Overview.doc

[7] ParaEngine and DHCI (paper).doc

[8] ParaEngine and WorldEditor.doc

[9] ParaEngine and VRML (unused).doc

[10] ParaEngine and Parallel World Game.doc

[11] ParaEngine and Reference Guide.doc

[12] ParaEngine and Frame Rate Control.doc

[13] ParaEngine and Game Engine Survey.doc

[14] ParaEngine and Physics.doc

[15] ParaEngine and Terrain.doc

[16] ParaEngine and Research Proposal.doc

[17] 2.0 ParaEngine and NPL.doc

[18] 2.1 ParaEngine and NPL module.doc

[19] 2.2 ParaEngine and NPL research.doc

[20] 2.3 ParaEngine and NPL in Game(paper).doc

[21] 2.4 ParaEngine and Scripting Reference.doc

[22] 2.5 ParaEngine and Script Tutorials.doc

[23] 2.6 ParaEngine and Scripting in Game (paper).doc

Paper Published During Bachelor Degree Study

[1] Xizhi Li and Qinming He, *Frame Rate Control in Distributed Game Engine*, *4th International Conference on Entertainment Computing*, 2005.

[2] Xizhi Li, *DHCI: an HCI Framework in Distributed Environment*, *11th International Conference on Human-Computer Interaction*, 2005.

[3] Xizhi Li, *Using Neural Parallel Language in Distributed Game World Composing*, *Distributed Framework of Multimedia Applications*, IEEE conf, 2005.

[4] Xizhi Li and Hao Lin, *A Proposal of Evaluation Framework for Higher Education*, *12th International Conference on Artificial Intelligence in Education*, AIED, 2005.

[5] Xizhi Li and Qinming He, *WAF: an Interface Web Agent Framework*, *International Conference on Information Technology*, 2004.

[6] Xizhi Li, *An HCI Template for Distributed Applications*, *International Conference on Computational Intelligence*, 2004.

[7] Xizhi Li and Tiecai Li, *ECOMIPS: An Economic MIPS CPU Design on FPGA.*, *IEEE International Workshop on System On Chips*, IEEE, 2004.

Acknowledgements

First of all, I wish to express sincere appreciation to my supervisor, Prof. He for his vast reserve of patience and knowledge. During my study at Zhejiang University, I have been given the maximum freedom in doing researches. I enjoyed the moment that Prof. He and I talked on my latest work.

I would also like to thank my family for their faith in me, specially my father. Father has given me the most encouragement and ideas. Unfortunately, in recent years, we work in different cities and can not meet as frequently as those early days. But when we meet, we will always carry on a lengthy and extremely stimulating talk.

I would also like to thank my fellow college mates: my partner WangTian, who is also my tennis coach, has helped me developing some of the code in ParaEngine; and my buddy LiLixuan, who is a great listener to my stories and a bold thinker himself. Recently, I have a small team of students helping me developing games based on ParaEngine. They are both hardworking and proactive.

During the past years, I have introduced my ideas to quite a few University professors as well as experienced game developers. They have given me valuable opinions. Especially I would like to thank Prof. Shu for his constant interest in my research, and Mr. Xu for his donation of electronic art resources to be used in our game demo.

Finally, I would like to thank my tutor Mr. Lu Yang, who has taught me how to play with and program in a computer since 1989 for ten consecutive years. None of my recent projects would have been possible without those early trainings and compassions.