Frame Rate Control in Distributed Game Engine

Xizhi Li
The CKC honors School of
Zhejiang University, P.R. China
LiXizhi@zju.edu.cn

Qinming He
Computer Science Department
Zhejiang University, P.R. China

Abstract:

Several new challenges arise in the visualization and simulation of distributed virtual environment of unsteady complexity. Time management (including time synchronization and frame rate control) is the backbone system that feedbacks on a number of optimization processes and computation modules to provide physically correct, interactive, stable and consistent rendering results. This paper proposes a unified time management (or more specifically frame rate control) architecture for distributed computer game engines. Several time management schemes are discussed, each of which is suitable for a specific game engine state or game requirement. The system has been used in our own distributed game engine and may also find applications in other simulation systems.

Key words: frame rate control, time synchronization, game engine

1 Introduction

Several new challenges arise in the visualization and simulation of distributed virtual environment of unsteady complexity. Time management (including time synchronization and frame rate control) is the backbone system that feedbacks on a number of optimization processes and computation modules to provide physically correct, interactive, stable and consistent rendering results. This paper proposes a unified time management (or more specifically frame rate control) architecture for distributed computer game engines. Several time management schemes are discussed, each of which is suitable for a specific game engine state or game requirement. The system has been used in our own distributed game engine and may also find applications in other simulation systems.

The big picture of time management in distributed game engines is this: animations and intelligent entities may adopt independent or master-slave time synchronization schemes; simulation and graphical computation routines are running at unstable (frame) rate; some portion of the dataset or game states is updated at variable-length intervals from multiple network servers; whereas the rest portion may be tailored using LOD (level-of-design) controls. In spite of all these things, a local game engine must be able to produce a single rendering frame rate, which best conveys the game state changes to the user (sometimes a physically correct animation under low frame rate is less satisfactory than time-scaled animation, which will be discussed in the paper). One way to solve the problem is to use different clocks for different engine modules and game objects, and use statistical or predictive measures to calculate the next time advance step. In other words, time management architecture should be carefully integrated to a computer game engine.

Current research on time management (including time synchronization and frame rate control) in the game context mainly focus on (1) variable frame length media encoding and transmission (2) time synchronization for distributed simulation (3) game state transmission in game server (4) LOD based interactive frame rates control for complex 3D environments. However, there have been relatively few literatures on a general architecture for time synchronization and frame rate control, which adapt to the more complex and variable situations in distributed computer game engines. In a typical game engine, modules that need time management support include: rendering engine, animation, I/O, physics engine, network engine (handling time delay and misordering from multiple servers), AI (path-finding, etc), script system, game video capturing system and various dynamic scene optimization processes such as ROAM based terrain generation, shadow generation and model LOD (level of details) control. Time management in the first four modules contributes most to the correctness and smoothness of the game play. Moreover, the time modules in these four modules usually form master-slave relationships with other modules in the game engine.

It is important to realize that timing in computer game is different from that of the reality and other simulation systems in a number of aspects. These include (1) interactivity or real-time game objects manipulation (2) correctness (e.g. physically correct motion) and consistency (e.g. consistent game states for different clients) (3) stable frame rate (this is different from interactive or average frame rate). For the above reasons, it is both irrational and unrealistic to synchronize all clocks in the entire networked virtual 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, due to the nature of current parallel architectures in both hardware and software (e.g. CPU and GPU; multi-threading, etc).

Instead of using ad hoc frame rate control programming, we suggest integrating our distributed time-management architecture to game engines. It is a both simple-to-use and flexible architecture for time synchronization and frame rate control. Its basic idea is to associate different kinds of clock (frame rate controller) to different objects in the game engine. Several kinds of relationships can be defined between clocks; the synchronization mode and frame rate control parameters can also be dynamically changed for these clocks during execution. For example, it supports equal-length interval mode, global synchronization mode, first-order interpolation mode, etc. The architecture has been used in our own distributed computer game engine (Li, 2004). The result is very satisfactory and will be demonstrated through the following time management configuration currently supported in our game engine: (1) interactive video capturing during game play (2) normal mode in standalone games (3) distributed multiplayer games.

In Section 2, we give the related works and a brief survey on the current implementations of time-related modules in computer game engines. In Section 3, we formulate the time management problem and propose our frame rate control architecture. In Section 4, we study some cases on time scheme configurations based on our own game engine. Section 5 concludes the paper.

2 Related Works and Surveys

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 distributed game server, game HCI, camera system, LOD, physical simulation, etc. Unfortunately, there are very few literatures on a unifying time management architecture for computer game engines. To our knowledge, there have been no open-source game engines which directly support time management so far. However, many questions have been asked in the game development forums, 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 can not be easily solved by a simple modification.

In this section, I will review the current implementations of a number of time-related modules in game engines. I have commented these methods so that people can figure out how to address them with our proposed frame rate control architecture, which will be presented following **Section** 2.

2.1 Decoupling Graphics and Computation

Several unsteady visualization environments have been developed which synchronize their computation and display cycles. These virtual reality systems separate the graphics and computation process, usually by distributing these functions among several platforms or system threads (multi-threading). Bryson's (Bryson, 1996) paper 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 for some game situations). 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.

I would like to point out here that 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 the game engine. 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 engine, however, we need to create more complex time dependencies between processes and also 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 reply on software or hardware parallelism to solve the problem.

2.2 I/O

The timing module for IO mainly deals with when and how frequently the engine should execute 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 (Bryson, 1996) 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 should be set to constant (i.e. independent of other frame rates).

2.3 Frame Rate and LOD

The largest number of related work (Funkhouser, 1993) (Xia, 1996) (Grabner, 2001) lies in Frame rate and LOD. However, our framework does not directly deal with how the LOD optimization module should react to feedbacks from the current frame rate; instead we are solely interested in how to provide such time feedbacks to modules not limited to LOD optimization.

2.3.1 Frame Rate and Scene Complexity

In many situations, a frame rate lower than 30 fps is also accepted by the user 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 (such as those mainly using a BSP scene manager), the camera is usually inside a closed room. Hence, the average scene complexity can be controlled fairly well by level designers. The uncontrolled part is mobile characters, which are usually rendered in relatively high poly models in modern games. However, scene complexity can still be controlled by limiting the number of high-poly characters in their movable regions, so the worst case polygon counts of any shot can stay below a predefined value. In multiplayer internet games, however, most character activities are performed in outdoor scenery which is often broader (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 can not afford to sustain a constant real-time frame rate (e.g. 30 FPS). A frame rate controller in game engine can ease such situations, by producing smooth animations even under low rendering frame rate.

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 (Cronin, 2002), a number of time synchronization mechanism for distributed game server are presented, with its own trailing state synchronization method. Diot (Diot, 1999) 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 (Lamport, 1978). This can either be done by preventing misordering outright (by waiting for all possible commands to arrive), or by having mechanisms in place to detect and correct misorderings. An additional complication to synchronization of multiplayer arises from their continuous nature. Even if there is no misorderings of commands from a single server, game state updates on the receiving client must be rescheduled for its visualization to be smooth. A third complication is that if a client is receiving commands from multiple servers, the time at which one command is executed in relation to other ones from other servers can create additional ordering constraints.

The ordering problem for a single server can usually be handled by designing new network protocols (which is both fast and detects misordering). Yet for the second and third problems, I personally do not think network protocol is a suitable place for handling time rescheduling of commands. For example, some distributed algorithm is able to synchronize a system of logical clocks which can be used to totally order events from multiple servers. However, what if our game requires that each game world has its own clock time and speed of time ticks? Hence, it is more suitable for game developers to use some time management scheme to solve these problems in the game engine, where some frame rate control architecture should be employed.

2.5 Physics Engine

The last category of related works I will discuss here is timing in physics engine, which is the trickiest part of all. (Out sourcing, 2002) provides a comprehensive overview about the time synchronization related problems with the use of a physics engine in game development.

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 (However, I will explain in Section 4, why this is NOT always true 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 systems.

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 frame time, we need to recalculate all the transforms from this bone to its root bone. Alternatively, we can use frame rate controller to achieve more flexible synchronization schemes. For example, we may allow one object's the physical parameters to be updated less frequently than the others. This may be the case, when the key framed animation is not a door or elevator, but a planet or mother spacecraft.

2.6 Conclusion to Related Works

In this section, I have enumerated 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, I believe it will soon become common in the backbone system of distributed computer game engines.

3 Frame Rate Control Architecture

In this section, I will propose the Frame Rate Control (FRC) architecture and show how it can be integrated in a modern computer game engine.

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) $\rightarrow$ {0,1}, where *t* is the time variable and f(t) is the frame time function. We associate a process in the game engine to a certain f(t) by the following rule: f(t) is 1, if and only if its associated process is being executed. The frame rate at time *t* is defined to be the number of times that the sign of f(t) changes from 0 to 1, during the interval (t-1,t).

Let $\{t^{s_k} \mid k \in N, \lim_{\delta x \to 0}(\frac{f(t^k) - f(t^k - \delta x)}{\delta x}) = +\infty\}$ 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. The curve of f(t) may be unpredictable in one way or another. See below.

(1) 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 rate is often of this type.

(2) 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.

(3) 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.

(4) 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 their time is independent and does not need much synchronization with other clocks.).

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. There are non-negotiable and negotiable constraints imposed to both t_n and f_n. 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. $(t_i - t_j) < \text{MaxFollowTime}, (\text{MaxFollowTime} > 0)$. One clock should follow another clock. The IO (user control such as camera move) and rendering 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* can not 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 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 successive 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 targeting on one hardware platform (such as PlayStation2 or Xbox) or 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-game video recording also needs this constraint to be applied to almost all game clocks.

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.

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. I will propose here the current integration implementation in our own game engine called ParaEngine (Li, 2004). 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 are used to set the current frame rate management scheme in the game engine. Each function will configure the frame rate controllers to some specific needs. 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 functioning under normal condition.

ParaEngine features to create distributed games so that each player can host their own virtual world on its computer. If we compare (1) web pages to 3D game worlds, (2) hyperlinks and services in web pages to active objects in 3D game worlds, and (3) web browsers and client/server side runtime environments to computer game engines, we will obtain the first rough picture of distributed Internet games targeted by our game engine. Yet unlike web pages each of which is displayed in a separate window (hence having their own reference of time), ParaEngine allows multiple game worlds on the network to be displayed and simulated (interacting) in a single scene. As an explanation, we can think of the following situation: the player was controlling its character walking near a village (imaging trees, animals and houses), when it saw a NPC (non-player character) talking with a third character. This is a typical game scene. However, what makes it interesting is that the village (trees, animals, houses) is simulated on computer A, whereas the NPC (which might possess high intelligence) is being simulated on an agent server B and the third character is coming from another client C. In the player's computer D, the game engine may need to hold connections to computer A, B and C in order to properly display the graphic and animations to its user. In distributed game engine, we do not have fixed client/server architecture, such as some central severs broadcasting game states to nearby clients. Hence, a new requirement to time management in distributed game engine is to integrate robust and distributed time management (frame rate control) to the entire engine framework. Figure 1 shows the architecture.

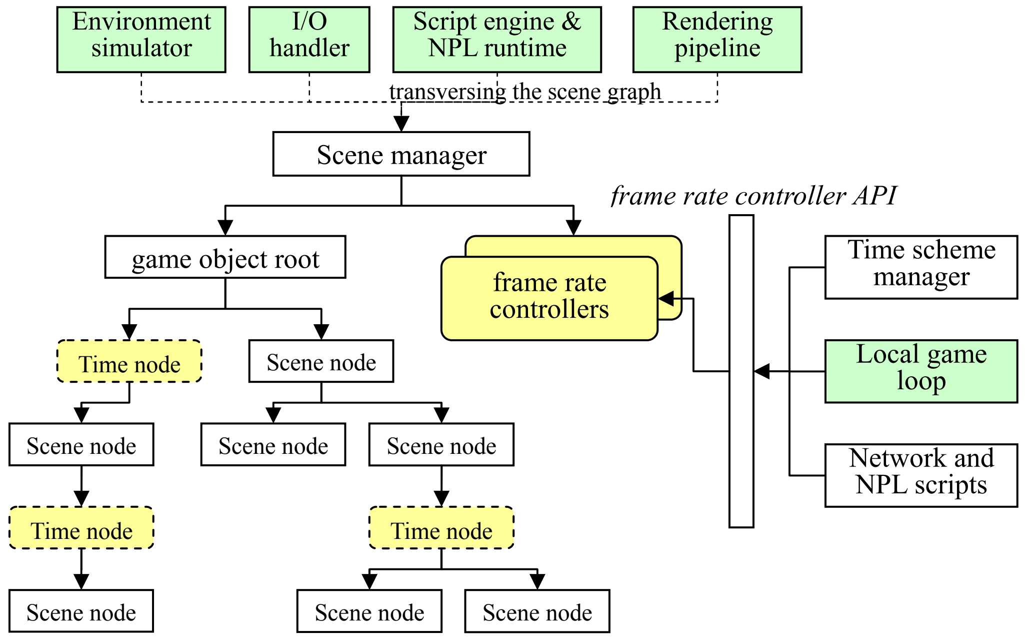

Figure 1. Integrating time control to the game engine

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:

(1) The FRC controllers inside *time* node will be applied to all its child scene nodes recursively.

(2) 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. Any changes made to the FRC controllers will be immediately reflected in the next scene traversal cycle.

4 Evaluation

This section contains some use cases of the proposed framework in our own game engine. 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. *Readers are welcome to download our game demo from the website (Li, 2004).*

4.1 Frame Rate Control in Video Capturing

The video system will create an AVI video while user is playing the game. When the video capture mode was 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 4FPS? Time management in the game engine can be complicated. 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, generates script events, performs accurate collision detection, runs environment simulation and plays 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 configurations for clocks used by its engine modules. In normal game play mode, we use N-scheme; whereas during video capturing, we use C-scheme. See **Table** 1.

Table 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

4.2 Coordinating Character Animations

In computer game engine, a character's animation is usually determined by the multiplication of its world transform and local transform. World transform determines the position and orientation of the character in the scene, which is usually calculated by the simulation engine. The local transform is usually directly computed from the animation data at current frame time. In order for the combined motion of the character to be natural, most developers would choose to coordinate the simulation time and local animation time with constraint (1) as given in section 3.1. However, this is not the best choice for biped animation for 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 generate more satisfactory result. This configuration does not generate strictly correct motion, but it does generate 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 has 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 smoothly under situation (i), but appears very jerky under situation (ii). This is because if simulation frame and local animation frame 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 intolerable jumpy to the user. However, with constraint (5), (6) applied, the local frame number render for the biped 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 are very smooth. The difference is that the biped will stride a larger step in situation (ii). But experiment shows that users tend to misperceive it as correct but slowed animation. This same scheme can be used for coordinating biped animations in distributed game world. For example, if there is any lag for a certain biped position update, we can automatically increase the stride of the biped, instead of playing jumpy animations with its original stride.

5 Conclusion

There are several new challenges for the visualization and simulation platforms of distributed virtual environment. Among them, time management is one of the most critical ones. The paper reviews a number of time-related modules in computer game engine and proposes a unified frame rate control architecture to be integrated in the backbone system of distributed game engines. The system has been successfully used in our own distributed game engine and may also find applications in other simulation systems.

Acknowledgement
ParaEngine is the distributed computer game engine developed by Xizhi Li. On top of ParaEngine, a distributed Internet game is currently being developed by a group of students at Zhejiang University.

Reference:

[1] C. Diot and L. Gautier. "A Distributed Architecture for MultiPlayer Interactive Applications on the Internet". In IEEE Network magazine, 13(4), August 1999.

[2] E. Cronin, B. Filstrup, and A. R. Kurc. A distributed multiplayer game server system. UM EECS589 Course Project Report, http://www.eecs.umich.edu/~bfilstru/quakefinal.pdf, May 2001.

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

[4] Thomas A. Funkhouser, Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. Computer Graphics (SIGGRAPH '93 Proceedings), volume 27, pages 247--254, August 1993.

[5] Xia, Julie, and A. Varshney. Dynamic View-Dependent Simplification for Polygonal Models. Proceedings of IEEE Visualization 96, 1996.

[6] S. Bryson and SandyJohan. Time management, simultaneity and time critical computation in interactive unsteady visualization environments. Published in proceedings IEEE Visualization '96, 1996

[7] Markus Grabner. Smooth High-quality Interactive Visualization. Proceeding of SCCG 2001, pages 139-148, April 2001.

[8] Lamport, L, Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558-565. 1978

The following content is for review purposes only.

ParaEngine Related Reference

http://www.lixizhi.net/publications.htm

[1] Xizhi Li. DHCI: an HCI Framework in Distributed Environment. 11th International Conference on Human-Computer Interaction (accepted for presentation)

[2] Xizhi Li. Using Neural Parallel Language in Distributed Game World Composing. In the Proceedings of IEEE Distributed Framework of Multimedia Applications 2005.

[3] Xizhi Li. Synthesizing Real-time Human Animation by Learning and Simulation. (to be submitted)

Some screen shots:

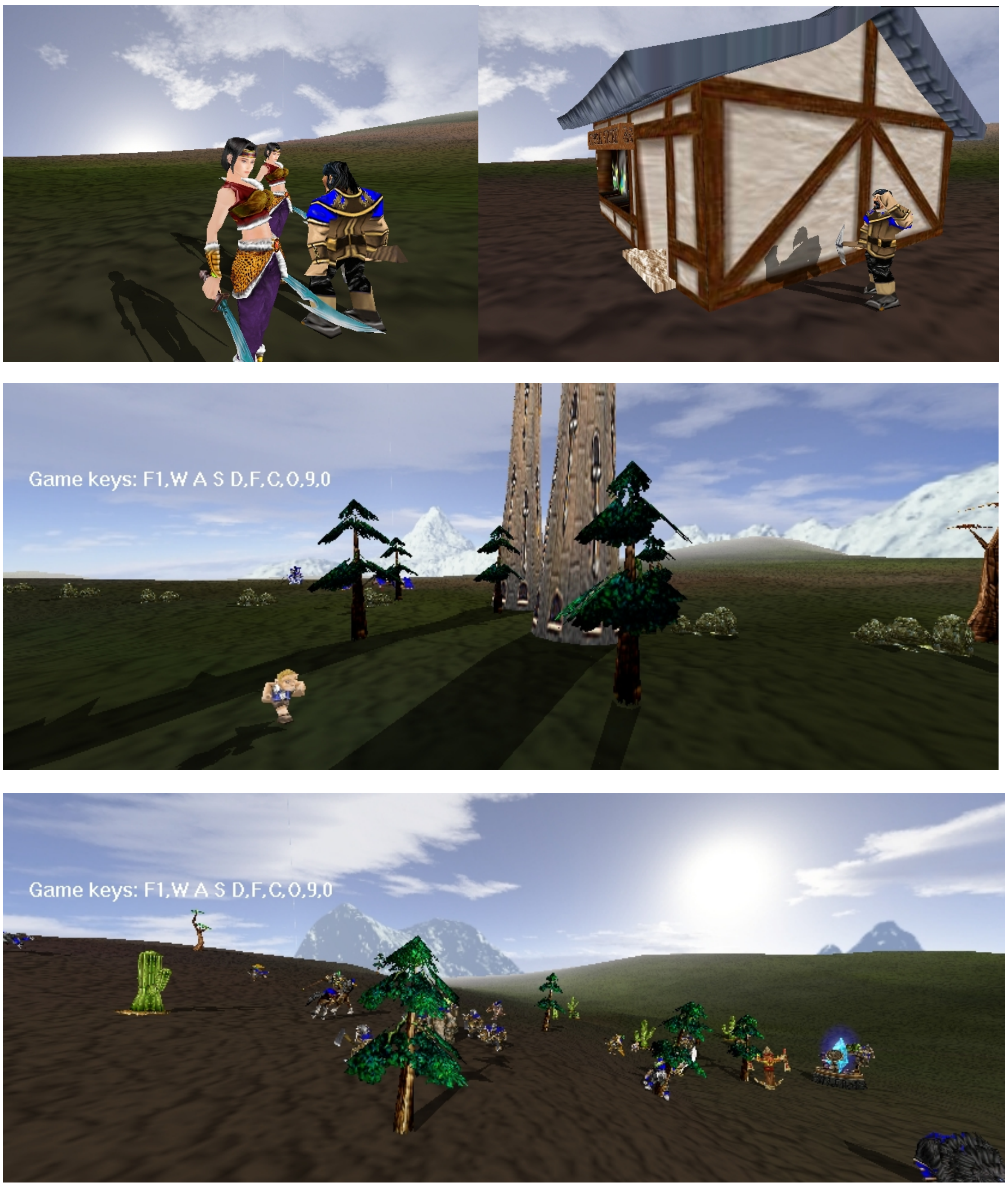
Game keys: F1,W A S D,F,C,0,9,0
Game keys: F1,W A S D,F,C,0,9,0

Figure A: coordinating animations and clocks among different scene objects and engine modules