Frame Rate Control in Distributed Game Engine

Xizhi Li[1], Qinming He[2]

[1] CKC honors School, College of Computer Science
Zhejiang University, Hangzhou, China 310027
lixizhi@zju.edu.cn
http://www.lixizhi.net
[2] College of Computer Science
Zhejiang University, Hangzhou, China 310027
hqm@cs.zju.edu.cn

Abstract. Time management (or 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. This paper discusses time related issues in game engines and proposes a unified time management (or more specifically frame rate control) architecture, which can be easily applied to existing game engines. The frame rate system has been used in our own distributed game engine and may also find applications in other multimedia simulation systems.

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.

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.

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 [6] 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.

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 [6] 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).

2.3 Frame Rate and LOD

The largest number of related work [4] [5] [7] 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.

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.

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 [2], a number of time synchronization mechanisms for distributed game server are presented, with its own trailing state synchronization method. Diot [1] 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 [8]. 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.

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 [3] 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.

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.

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.

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.

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. Fig. 1 shows the curves of three related frame rate functions: i, j and k.

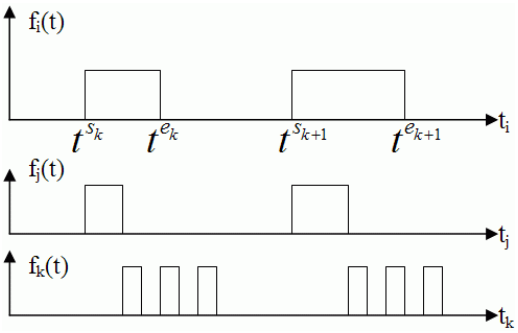

Fig. 1. Sample curves of frame rate functions

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.

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 our own game engine called ParaEngine [9]. ParaEngine is an experimental game engine framework aiming to bring interactive networked virtual environment to the Internet through game technologies.

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 Fig. 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 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 Fig. 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.

Table 1. The Game Loop of ParaEngine [9], which drives the flow of the engine, providing a heartbeat for synchronizing object motion and rendering of frames.

```
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)
}
```

Since frame rate controllers are managed as top-layer (global) objects in the engine (see Fig. 2). Any changes made to the FRC controllers will be immediately reflected

in the next scene traversal cycle. Table 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.

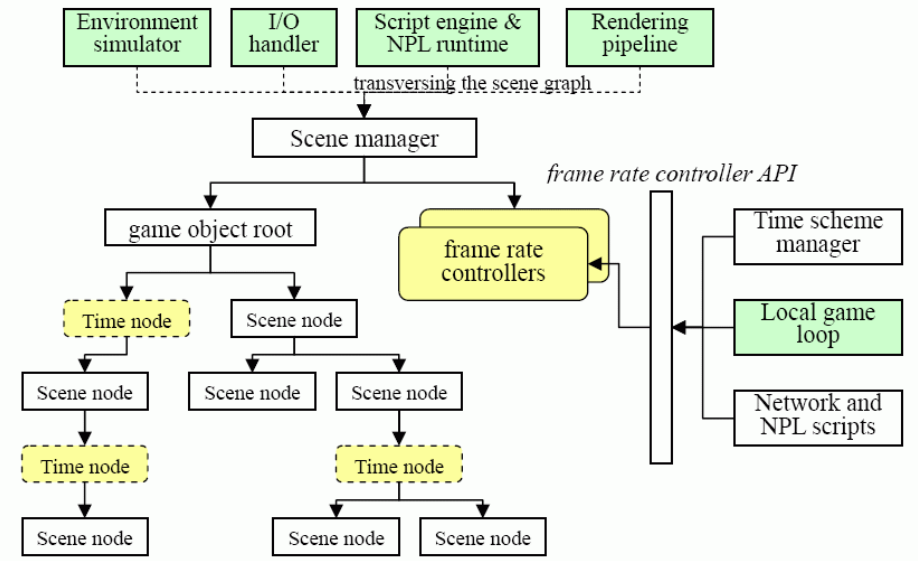

Fig. 2. Integrating time control to the game engine

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 links in the thesis [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 2. 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 2. 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 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.

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.

References

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), vol 27, 247--254, Aug. 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
9. Xizhi Li.: Distributed Computer Game Engine – research and implementation. B.C. thesis, Zhejiang Univ. http://www.lixizhi.net/paraworld/DisGameEngineThesis.pdf, June 2005