Federated-Distributed Simulation of Rigid Bodies in Computer Games

Pawan Kumar and Qasim Mehdi

School of Computing and Information Technology
University of Wolverhampton
Wolverhampton, UK WV1 1SB
{pawan.kumar, q.h.mehdi}@wlv.ac.uk

KEYWORDS

Rigid Bodies, Distributed Simulations, FDK, OpenGL, Interest Management

ABSTRACT

In this paper we detail our prototype rigid bodies simulation environment that was developed for testing and evaluation of federated simulation development kit (FDK) in computer games. Also we discuss the algorithms that were developed for rigid bodies simulation including the dynamic collision detection, numerical methods and discuss the approach used for distributed rigid bodies using FDK.

1. INTRODUCTION

Recent advances in broadband and Internet technologies has led to better and advanced online game play features with support for hundreds and thousands of simultaneous simulation nodes. However, even with these advancements, the requirements of the game often exceeds the available technology support and thus demands efficient and scalable distribution algorithms on top of the existing communication medium. Such schemes are termed as Interest Management schemes and are incorporated in networking middleware systems to allow for scalability that will support large number of simulation nodes efficiently and seamlessly. These schemes aim at minimising inherent latencies and conservation of bandwidth by reducing information exchange across the simulation nodes.

Interest management systems have been used in several large-scale distributed simulators [2,3], collaborative virtual environments [4,5,6] and multiplayer online games [7,8,9]. In online games, a player's node will contain some subset of the shared virtual world whose state is influenced and maintained by the player. In order to have a mutual consistent view of the virtual world, events are exchanged between player and other simulation nodes (either directly or indirectly through a server) [1]. However, an update occurring at one node is likely to have an immediate significance on only a subset of nodes in the system. It is this feature that is exploited by most of the interest management techniques.

Much of our previous research has focused on scalable algorithms for Interest Management in online games [10,11] that were developed for federated simulation development kit (FDK) [12]. In order to evaluate the effectiveness of the algorithms and test its usage in FDK, a prototype game scenario involving distributed simulation of rigid bodies was created in this work. In this paper we detail the game scenario, the algorithms implemented for simulation updates, collision detection, rendering and approach used for distributing the rigid bodies simulation over FDK.

2. DISTRIBUTED SIMULATION OF RIGID BODIES

The game scenario involves rigid bodies and is based on a virtual 3D air hockey game simulation (Figure 1). The game comprises of a rectangular playing field with two goals at opposite ends with a maximum of two players playing on the field. The playing field is divided into two halves (one for each player) and there are two cylindrical posts in the middle of it (one on each half). The two players have a rectangular bat and cylindrical pucks to play with. To win the game, each player is required to score at the opponent's goal. The player defeats another player when he scores a maximum of five goals. Each player has an associated keyboard controller to control the bat. The bat can be forced to move left, right, up and down to hit the puck into opponent's goal. In distributed version of the game, each player has its own client area that renders the game from his perspective.

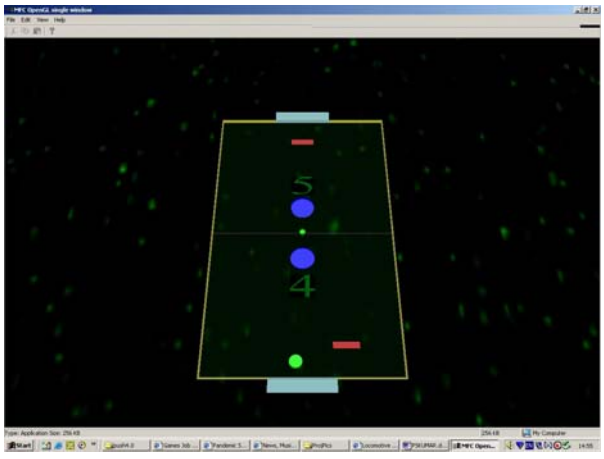

Figure 1 Virtual Air Hockey Simulation

The development of the game scenario was carried out using C++, visual studio .net 2005 and OpenGL.

OpenGL provides a standard, clean and user-friendly interface for graphics programming whereas visual studio .net environment provides ample tools for debugging and software engineering. An open source OpenGL framework GLFW [13] is used for rapid prototyping of the game across multiple platforms. The GLFW framework provides platform independent windowing system with registration of keyboard and mouse callbacks, platform independent multi-threading support with thread safe OpenGL rendering context. In addition, the simple integration of GLFW with FDK allows choosing GLFW framework from the pool of other similar open source OpenGL frameworks.

The design of the game scenario at the architecture level follows the proven Model-View-Controller [14] pattern that decouples all game models from their views and controllers (Figure 2). The abstract classes CModel and CController are provided so that concrete game models and controllers can be inherited from them. The CGame class acts as a one-stop shop for creation-deletion (factory pattern), and overall rendering and updating of the game. The CGame class also acts as a mediator (mediator pattern) between all the game models so that each game model can query the game state about other models using the CGame class interface. Further, CGame singleton instance is the only instance that the GLFW framework requires to initialise the game states and set the game running.

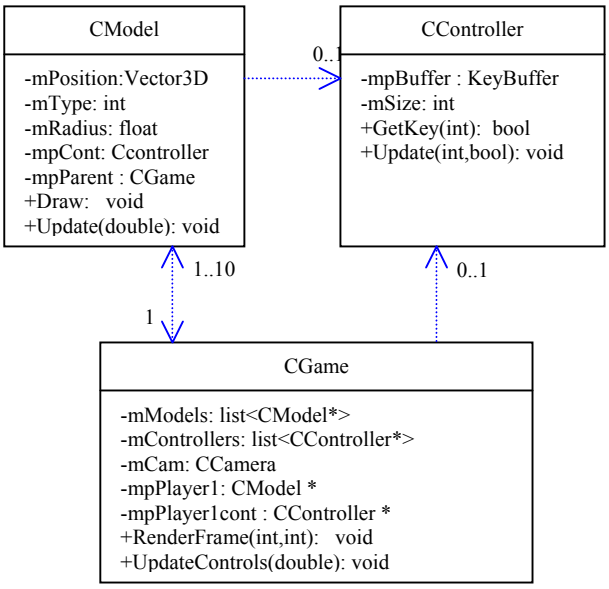

Figure 2. Class Design (Architecture Level)

In addition to the architecture level class design, there are two sets of parallel inheritance hierarchies for models and controllers in the game (Figure 3). Moreover, several other utility classes such as vector, plane, rectangular block, texture loader and camera were created to realize the games graphics and simulation algorithms. The pitch and bats in the demo were made up of planes and were rendered as GL_QUADS. For cylindrical posts and pucks, the GLUquadricObj instance is used for visual representation. The background box and scores were implemented using textured box and textured planes respectively as depicted in Figure 1. In the following subsection we detail the simulation algorithm with particular emphasis on the collisions and physics.

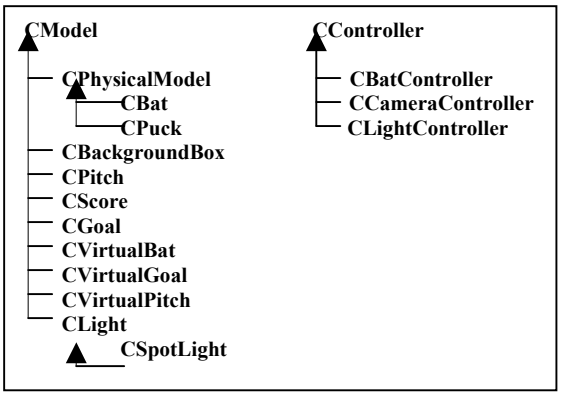

Figure 3. Game's Inheritance Hierarchy

2.1. Collision System and Simulation Updates

The core of the simulation of rigid bodies lies at the collision detection algorithms, numerical methods and the physically based responses. Several approaches to simulate rigid bodies have been established in both academic research and commercial games [15,16]. These vary from using Newtonian physics and numerical integration for positional updates, collision detection algorithms using static and dynamic (time based) approaches, simulation updates using conservative or optimistic approaches, to hacking and tweaking of crude algorithms to get the desired effect. Our simulation of rigid bodies was aimed at realistic physical updates with proper collision responses that reflect Newtonian principles. For this we developed our collision library that takes into account the exact time of collision between two colliding bodies. Figure 4 and 5 shows the scenario from static and dynamic (time based) collision detection perspective respectively.

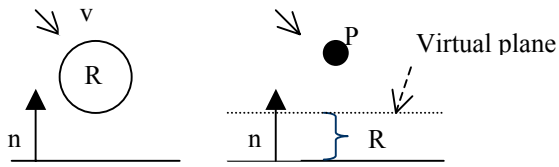

Figure 4. Collision Scenario Between Puck and Wall

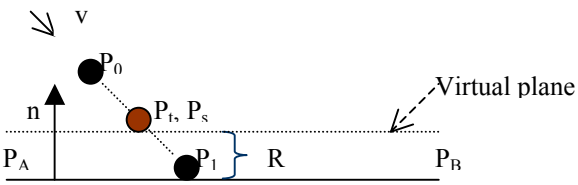

Figure 5. Dynamic Collision Detection Scenario

In Figure 4, the left part shows the game scenario where the puck of radius R is moving in the direction

of vector v and is approaching towards a wall plane that has its normal in direction n. On the right, the same is converted into a static collision problem where point P replaces the puck and a new virtual plane is created parallel and inwards to the original wall plane at a distance R (equal to the radius of the puck). Here the problem is *static* as there is no consideration of time and the test involves calculating the distance of the *static point P from the static virtual plane* to determine whether the puck is in collision with plane or not. However, for proper physical response it is imperative to consider the puck's direction of travel along with exact time of collision. This is shown in Figure 5 where puck's position at current frame is P_0 and its position at the next frame is P_1. Clearly at current frame the static test would yield no collision while on the next frame, the static test would result in collision but the puck would have already penetrated the wall. In order to overcome this scenario, a more dynamic approach is required where proper instance or time of collision is considered while updating the puck's position. This requires reformulating the original problem to a *moving point-static plane* test. To achieve this *a parametric equation of line* is used to represent moving point Pt such that:

$$\mathbf{P_t = P_0 + t * (P_1 – P_0): t>=0 \text{ and } t<=1}$$

Further, the static virtual plane is also represented by parametric equation of line given by two end points P_A and P_B (Figure 5). Thus, any point P_s on the virtual plane can be obtained as follows:

$$\mathbf{P_s = P_0 + s * (P_1 – P_0): s>=0 \text{ and } s<=1}$$

Given the two equations, we find the collision point where P_s equals P_t. Equating and solving for either one of the parameters (s or t) results in finding the exact point or time of collision to give appropriate responses.

The above approach holds good only for getting exact collision detection between the puck and the virtual plane. However, the same cannot be used for the puck and the rectangular shaped player bats (Figure 6). The Player's bat is a movable entity and instead of sweeping a line (moving point), it sweeps an area (a plane). For exact time based collision detection between the puck and the bat, we formulated it as a *moving point-moving edge* problem and did the test with the four edges that comprise the virtual bat (virtual bat planes are offset from the bat planes by the radius of the puck). For each of the bat planes, a moving edge is represented by a parametric equation of plane that is equated with the moving point equation of the puck to find the exact time of collision.

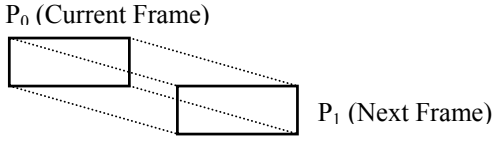

Figure 6. An Update of Rectangular Bat Sweeps an Area.

In order to implement the collision detection in our game, we implemented a generic collision detection library with functions for calculating both the static and dynamic collision detection. In addition to point-edge checks, several other collision checks such as *moving circle-static circle* (for puck-post collision), *moving circle-moving circle* (for puck-puck collision), *point in circle and bounding circle* were implemented to realize the game's collision requirements.

The rigid bodies updates in the game were realized based on Newtonian second laws of motion where the position and velocity is updated based on the current accumulated force. These updates were realized based on standard Euler integration method [16]. Drag forces in the direction opposite to the current velocity is added to simulate the effect of friction and to minimise the energy gained in the system due to errors in Euler Integration. Further, for collision checks, we take a two-tier approach. First, a simple static check is made to see if the two entities are in close proximity to each other, and once the static test succeeds, the more complex dynamic tests are carried out to find the time of collisions. In addition, to realistically simulate the movements, we adopted optimistic approach to simulation where the current states of the entity (position and velocity) are saved before updates. Once a dynamic collision is detected, we get the exact time of collision and the simulation is then rolled back to the start of the time frame, updated again to the time of the collision, apply appropriate responses at that time and finally update the entities with the remaining time. Such considerations were imperative for realistic collision responses especially when multiple bodies are in close proximity. The collision response in our demo follows an impulse-momentum based approach where principles of conservation of momentum is used to give appropriate responses to the colliding bodies. In the following section 2.2, we discuss the implementation of federated simulation of the rigid bodies.

2.2. Distributed Rigid Bodies Using FDK

Distributing any virtual environment or gaming simulation across multiple nodes requires addressing the issues of selecting proper networking architecture, choosing efficient state distribution strategies, using proper interest management schemes for scalability and information culling and lastly looking at impact of balanced-unbalanced work load across the nodes [1]. Our work uses FDK that provides run-time infrastructure (RTI) for federated simulation. From the games perspective, FDK offers an ideal platform for research in online games as it provides a set of reusable libraries and allows games researchers to work at three levels of abstraction. Firstly, *application developers* can use the FDK API for developing high-performance distributed simulations using FDK run-time. Secondly, *communication developers* can configure and realize multiple instances of the software kit for several different target platforms such as shared memory multiprocessors, cluster computing environments, local

and wide area connected workstations that use standard networking protocols such as TCP/IP. They can also extend and develop their own communication protocols and messaging system using higher-level communication interfaces. Finally, for *RTI developers*, the kit offers modules and utilities such as basic time management services, data distribution management services, group communication management, buffering, etc. Thus, this modular approach allows researchers and developers to utilize and add features to the kit depending upon the application and target platform. Our previous work of interest management algorithms [10,11] focussed at the RTI level where we presented scalable algorithms for data distribution in the FDK. This work uses the FDK at the application level for distributing the rigid bodies game across the player nodes.

Networking architectures such as client-server, peer-to-peer and hybrid [8] play an important role in multiplayer games and selecting a proper architecture is an important engineering decision so that a consistent game state is maintained and viewed by all the players. For rigid bodies, we experimented with the client-server architecture. We set up federation execution by spawning 3 nodes. Two nodes acted as client (player) nodes that were responsible for rendering the game and updating the bat controllers. The third node (server) maintained the complete game state and carried out simulation updates of the rigid bodies. For communication, the FDK default synchronous execution environment over TCP is used and the game is played on the LAN. All the nodes were spawn on Pentium 4 machine running Windows operating system.

Since all the persistent shared-state is maintained at the server, we don't use any interest management features of the FDK (this work will be used for evaluation purposes in the future). Further, since FDK does not allow for dynamic discovery of the nodes, the details of all the nodes are written in the script file that is parsed by the application during the initial setup. At the time of setting up the federation execution, each of the nodes (either client or server) enters a barrier and stalls the execution until all other nodes have properly initialised and reached their respective barriers. Once that step is reached, information can be exchanged among the nodes using the API provided by the FDK.

The information exchange between client and server is specified using meta-level description in a script file, which is then parsed at the application load time and subsequently, meta-objects are instantiated. These meta-objects are related to objects in the game and stand in as proxies to relay information between clients and server using the FDK. Further, these meta-objects are registered as observers (observer pattern) of in game objects. Whenever a game object updates its state, its observers are notified so that they can reflect these changes. This approach is in line with our Observer of Observer design for integrating FDK with other game engine that we used in our previous work [17].

FDK offers two types of meta-level objects. The first are object classes that allow publishing-subscribing the entire class or its attributes. The second are interaction classes that allow publishing-subscribing to the interactions between the nodes. For our work, we made use of both these classes. An *Entity* object class with an attribute of *position* is used for retrieving the positions from the server. A *Score* class is also used to retrieve player scores from the server. Moreover, clients use an *Update* interaction class to send notification to server to update the game state. Server subscribes to *Update* interaction class to receive notifications from clients. Server uses its own internal timer to update all the game states and storing the updated positions and score information in their respective object classes. Clients also use interaction classes for sending the players key controller information. Instead of sending force vector (3 floats) to reflect changes in the position of the bat, each client sends key state (char) as an interaction class. Two types of interaction classes (*KeyUp* and *KeyDown*) were used for sending key state information.

3. CONCLUSIONS AND FUTURE WORK

In this paper we presented one approach for distributed simulation of rigid bodies using the FDK kit. Our approach for realizing rigid bodies was mainly driven by realistic physical simulation and we presented the techniques used for implementing that. We detailed the use of static and dynamic collision detection approaches during the simulation updates. Further, we detailed the use of numerical integration with optimistic simulation update and collision detection for finding the exact time of collision so that appropriate reposes can be given to the colliding bodies. The techniques used resulted in a very impressive demo of the game. We experimented with the client-server architecture where server maintained and updated the complete game state and highlighted how FDK is used and integrated with the rigid bodies demo to make it a multiplayer game.

Our experimentation of distributing rigid bodies with the FDK kit is first step towards evaluation of Interest management algorithms that we developed for FDK data distribution management service. The intent of this work was on physically realistic simulation of rigid bodies because it is a computationally expensive problem. The use of FDK would allow for exploiting efficient parallelization and distribution strategies for such compute intensive problems. In future, we will investigate other distribution architecture such as peer-to-peer where the game state would be shared across clients rather than maintaining on the server. In addition, we will populate the game world with several instances of bats and pucks and change the game play to evaluate the scalability and effectiveness of the interest management algorithms.

References

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

[2] Morse K. 2000. "An Adaptive, Distributed Algorithm for Interest Management"; *PhD Thesis*, University of California, Irvine

[3] US Defence Modelling and Simulation Office. 1998. High Level Architecture (HLA)- Interface Specification, version 1.3

[4] Macedonia M, Zyda M, Pratt D, Brutzmann D and Barham P. 1995. "Exploiting Reality with Multicast Groups: A Network Architecture for Large-Scale Virtual Environments"; *IEEE Computer Graphics and Applications,* 15(3): 38-45

[5] Miller D and Thorpe J A. 1995. "SIMNET: The Advent of Simulator Networking", *Proc. of IEEE,* 83(8): 1114-1123

[6] Greenhalgh C and Bendford S. 1995. "MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading", *Proc. of 15th International conference on distributed computing systems (DCS 95)*, IEEE Computer Society, 27-35

[7] Epic Games 1999. *The Unreal Networking Architecture*. World Wide Web, http://unreal.epicgames.com/Network.htm

[8] Yu A and Vuong S T. 2005. "MOPAR: A Mobile Peer-to-Peer Overlay Architecture for Interest Management of Massively Multiplayer Online Games", *in proc. of International Workshop on Network and Operating systems Support for Digital Audio and Video,* pp: 99-104

[9] Liu E, Yip M and Yu G. 2005. "Scalable Interest Management for Multidimensional Routing Space", *in proc. of the ACM symposium on Virtual Reality Software and Technology,* pp: 82-85

[10] Kumar P and Mehdi Q. 2006. "Recursive Interest Management for Online Games", *in proc. of 8th International Conference on Computer Games (CGAMES),* Louisville KY, USA.

[11] Gallego F, Kumar P, Mehdi Q and Llorens F. 2007. "Constant-Time Complexity Interest Management for Online Games", *in proc. of 10th International Conference on Computer Games (CGAMES),* Louisville KY, USA.

[12] FDK- Federated Simulations development kit. Available: http://www.cc.gatech.edu/computing/pads/software.html

[13] GLFW. Available: http://glfw.sourceforge.net/

[14] Gamma E et al 1995. *Design patterns: elements of reusable object-oriented software.* Addison Wesley, 1995

[15] Baraff D and Witkin A. 1997. "Physically Based Modelling: Principles and Practice". Online Siggraph Course Notes. Available: http://www.cs.cmu.edu/~baraff/sigcourse/

[16] Eberly D. 2004. *Game Physics.* Morgan Kaufmann Publishers 2004

[17] Kumar P. 2005. "Towards Integrating Ogre3D with FDK", *in proc. of 7th International Conference on Computer Games (CGAMES),* Angouleme, France