Chapter 2
A Distributed Architecture for Simulation Environments Based on Game Engine Systems

Mark Joselli, Marcelo Zamith, Luis Valente, Bruno Feijó, Fabiana R. Leta and Esteban Clua

Abstract Simulation systems are becoming common in different knowledge fields, such as aeronautics, defense, and industrial applications, among many others. While in the past these systems where mostly based on typical Virtual Reality Environments, with the advance of the game industry simulators are being developed using typical game engines and gaming software architectures. Distributed computing is being used in several fields to solve many computation intensive problems. Due the complexity of Simulation systems, this architecture can also be used, devoting host processing to renderization, which is usually the task that simulators spend most of its processing time. By using distributed computing, simulators could need softer system requirements, since the main loop would be distributed. This work presents concepts of simulator software, which is based on the main loop technique. After describing state-of-the-art concepts, we present an efficient automatic load balancing and distributing logic computation among several computers for simulators.

M. Joselli · M. Zamith · L. Valente · E. Clua (✉)
Computing Institute, Universidade Federal Fluminense-UFF, Niterói, RJ, Brazil
e-mail: esteban@ic.uff.br

M. Joselli
e-mail: mjoselli@ic.uff.br

M. Zamith
e-mail: mzamith@ic.uff.br

L. Valente
e-mail: lvalente@inf.puc-rio.br

B. Feijó
Informatic Department, PUC-Rio, ICAD Games, Rio de Janeiro, RJ, Brazil
e-mail: bfeijo@inf.puc-rio.br

F. R. Leta
Mechanical Engineering Department, Universidade Federal Fluminense-UFF, Niterói, RJ, Brazil
e-mail: fabianaleta@id.uff.br

F. R. Leta (ed.), *Visual Computing*, Augmented Vision and Reality 4,
DOI: 10.1007/978-3-642-55131-4_2, © Springer-Verlag Berlin Heidelberg 2014

Keywords Simulation · Real-time visualization · Game engine architecture · Task distribution

2.1 Introduction

Increasing realism level in virtual simulations depends not only on the enhancement of modeling and rendering effects, but also on the improvement of different aspects such as animation, artificial intelligence of the characters, and physics simulation. Real-time systems are defined as solutions that have time constraints to run their tasks. Hence, if the system is unable to execute its work under some time threshold, it will fail. In order to achieve such constraints, the main loops have to be carefully implemented. The main loop is the design pattern of such kind of applications.

Real-time simulators are applications that employ knowledge of many different fields, such as computer graphics, artificial intelligence, physics, computer networks, and others. While these are typical requirements found in games, simulations usually require these features with much more accuracy. More, computer simulators are also interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition is related to gathering data from input devices as keyboards, mice, and dedicated interfaces, depending on the simulator. Data processing tasks consists on applying logic rules, responding to user commands, simulating physics, and artificial intelligence behaviors. Data presentation tasks relate to providing feedback to the user about the current simulation state, usually through images and audio. Many simulators are included in multiuser environments, requiring the usage of distribution and logical partitioning of the scene [1].

Simulators are interactive real-time systems and have time constraints to execute all of their processes and to present the results to the user. If the system is unable to do its work in real time, it will lose its interactivity and consequently it will fail. A common parameter for measuring the performance is frames per second (FPS). The general lower acceptable bound for a game is 16 FPS. There are no higher bounds for a FPS measurements, but when the refresh rate of the video output (a computer monitor) is inferior to the game application refresh rate, some generated frames will not be presented to the user (they will be lost). One motivation for designing loops optimizations is to better achieve an optimal FPS rate for the application. Doing so, it is possible to spend more time with higher precision physical calculation or more complex logic behaviors.

The architecture that we present in this paper follows a similar concept as cloud and distributed computing, where machines across the Internet shares resources, software, and information, where the user's computer can use other resources available on the network, to help it in processing the application. By using this approach, a computer with less computing power could join the simulation session, by relaying the effort to process the system to the network cloud.

This work summarizes several works of the authors, more specially [2–6], which presented different approaches for automatic task distribution between CPU cores and GPU in game and simulation environments, for single or multithreaded loops. In this work, we extend and describe more details related to the distribution of tasks of main loop and a smart load/balancing of the tasks.

This chapter is organized as follows: Sect. 2.2 presents a set of real-time loop model concepts found in the literature, specially coming from the gaming field. Section 2.3 presents a framework architecture concept that can be used for multiplatform simulation environments, with dynamic load balancing. Finally we present the conclusions.

2.2 Related Works

The real-time loop is the underlying structure games and real-time simulations are built upon. These loops are regarded as real time due to the time constraints to run the game-related tasks. This loop may become a bottleneck in complex simulations, due the complexity of physics and visualization requirements.

As mentioned earlier, the tasks that a computer simulation should execute can be broken down into three general groups: data acquisition, data processing, and presentation. Data acquisition means gathering data from available input devices, such as, mice, joysticks, keyboards, motion sensors, and dedicated input devices, such as an aircraft panel or car instrumentation. The data processing part refers to applying the user input into the application (user commands), applying simulation rules (the simulation logic), simulating the world physics, the artificial intelligence, and related tasks. The presentation refers to providing feedback to the user about the current simulation state, through images, audio, and in some cases motion.

Real-time simulators provide the illusion that everything is happening at once. Since these systems are interactive applications, the user will not have a good experience if the systems are not able to deliver its work on time. This issue characterizes these systems as heavy real-time applications. Although the real-time loop represents the heart of real-time simulations, it is not easy to find academic works specifically devoted to this subject. The works by Valente et al. [7], Dalmau [8], Dickinson [9], Watte [10], Gabb and Lake [11], and Monkkonen [12] are among the few ones.

The simplest real-time loop models are the coupled ones. The Simple Coupled Model [13] is perhaps the most straightforward approach to modeling real-time loops. It consists of sequentially arranging the tasks in a main loop. Figure 2.1 depicts this model.

The uncoupled models separate the rendering and update stages, so they can run independently, in theory. These models consider single-thread [9, 13] and multithread designs [11–13]. The Multithread Uncoupled Model [13] and the Single-thread Uncoupled Model [13] try to bring determinism to the simulator execution

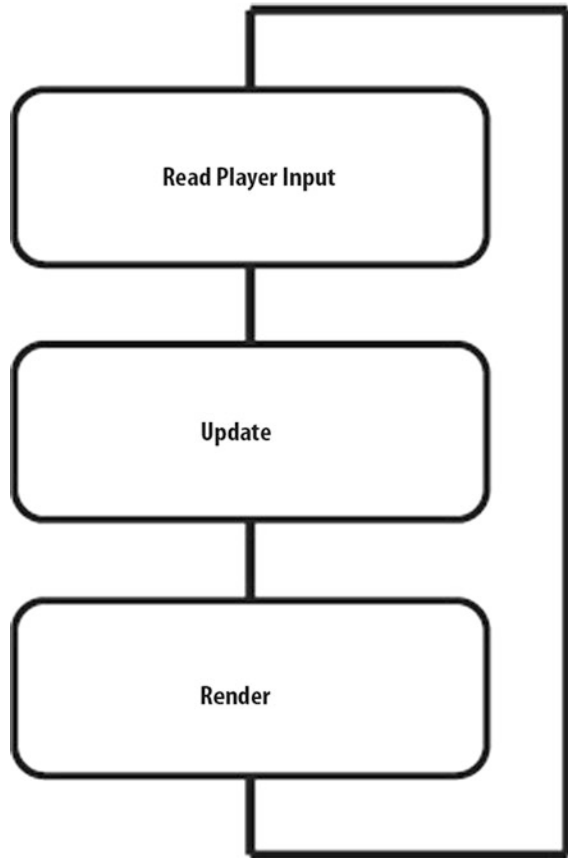


Fig. 2.1 Simple coupled model

by feeding the update stage with a time parameter. Figures 2.2 and 2.3 illustrate these models, respectively.

By using these models, the application has a chance to adjust its execution with time, so the system can run the same way in different machines. More powerful machines will be able to run the simulation more smoothly, while less powerful ones will still be able to provide some experience to the user. This requirement is also important due the fact that hardware configuration change fast, while the simulator may have a longer time of life.

Although these are working solutions, time measuring may vary greatly in different machines due to many reasons (such as process load), making it difficult do reproduce it faithfully. For example, some training sessions may require a scene replay feature [5], which may not be trivial to implement if it is not possible to run some part of the loop sequence in a deterministic way. Other features as network module implementation and program debugging [5] may be easier to implement if the loop uses a deterministic model. Another issue is that running some simulations too frequently, like AI and the game logic, may not yield better results.

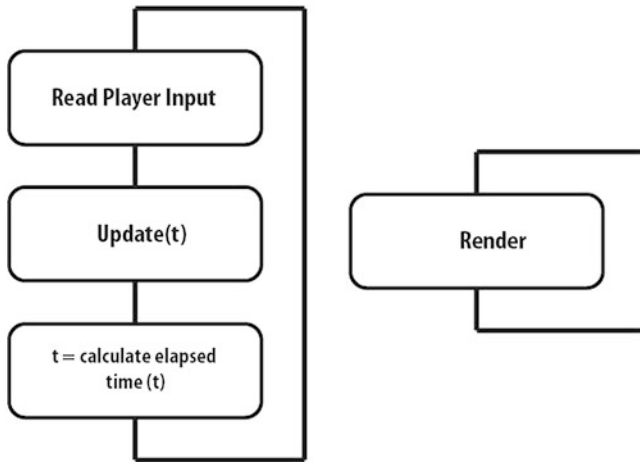


Fig. 2.2 Multithread uncoupled model

Hence, the models proposed in [9, 13, 14] try to address these issues. The Fixed-Frequency Uncoupled Model outlined in [13] features another update stage that runs at a fixed frequency, besides the time-based one. Dalmau [8] presents a similar model, although not naming it explicitly. These works describe the model using a single-thread approach. Figure 2.4 illustrates the Fixed-Frequency Uncoupled Model.

The model described in [9] presents just one update stage that runs at a fixed-frequency, whose main objective is to attain reproducibility. Another interesting model is the one used in the Microsoft XNA framework [14]. The XNA model has an update stage that runs at a fixed frequency or freely, but not both. The user is able to set a parameter that informs the XNA framework about which one to use. While this is an old framework, modern engines still keep using the same approach. This is especially relevant for simulators, since they typically use high profile machines.

Dealing with concurrent programming introduces another set of problems, such as data sharing, data synchronization, and deadlocks. Also, as Gabb and Lake [11] states, that not all tasks can be fully parallelized due to dependencies among them. As examples, a dynamic simulation element cannot move until the complete logic is computed, and visualization cannot be performed until the game state is updated. Hence, serial tasks represent a bottleneck to parallelizing simulation computation.

Monkkonen [12] presents models regarding multithread architectures that are grouped into two categories: function parallel models and data parallel models. The first category is devoted to models that present concurrent tasks, while the second one tries to find data that can be processed entirely in parallel. The Synchronous Function Parallel Model [12] proposes to allocate a thread to all tasks that are (theoretically) independent of each other. For example, performing complex physics simulation while calculating animation. Figure 2.5 illustrates this model.

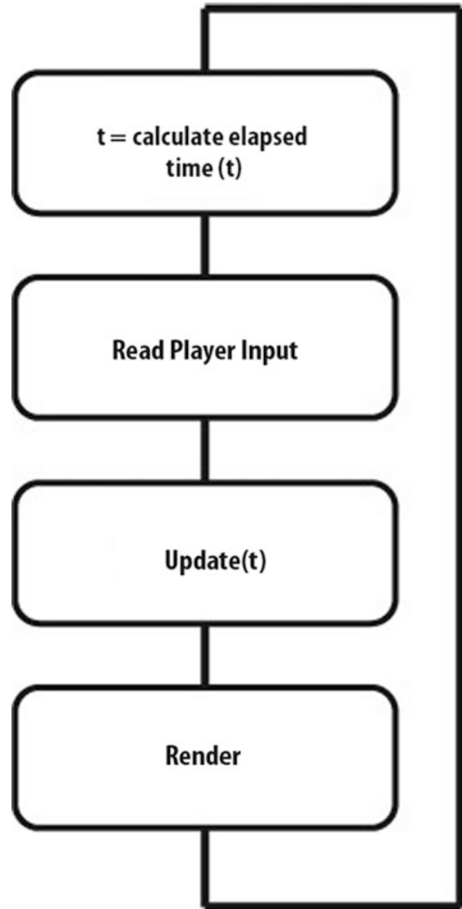


Fig. 2.3 Single-thread uncoupled model

Monkkonen states that this model is limited by the amount of available processing cores, and the parallel task should have little dependency on each other. The Asynchronous Function Parallel Model [12] is the formalization of the idea found in [11]. This model does not present a main loop. Figure 2.6 illustrates the model.

Different threads run the simulation tasks by themselves. The model is categorized as asynchronous because the tasks do not wait for the completion of other ones to perform their job. Instead, the tasks use the latest computed result to continue processing. For example, the visualization task would use the latest completed physics information to draw the objects. This measure decreases the dependency among tasks. However, task execution should be carefully scheduled for this scheme to work nicely. Unfortunately, this is often out of the scope of the application. Also, serial parts of the application (like rendering) may limit the performance of parallel tasks [11].

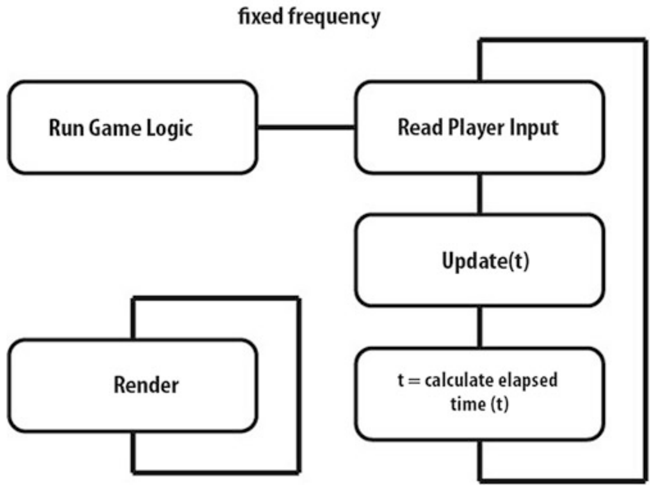


Fig. 2.4 Fixed-frequency uncoupled model

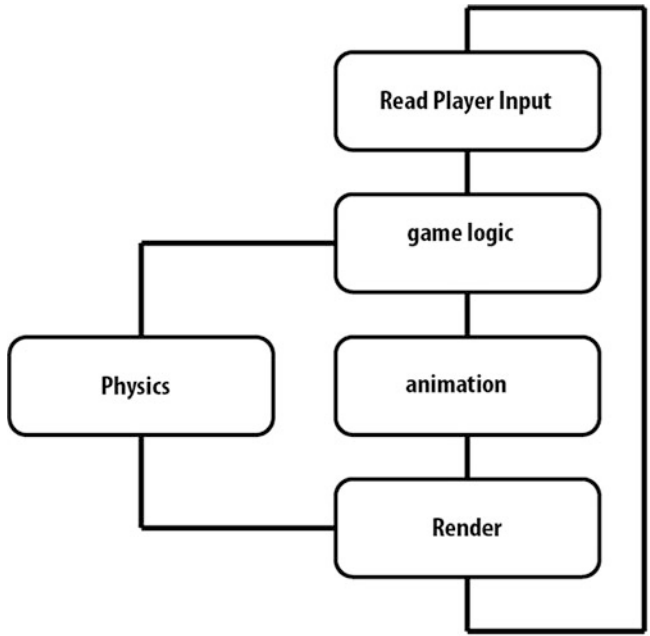


Fig. 2.5 Synchronous function parallel model

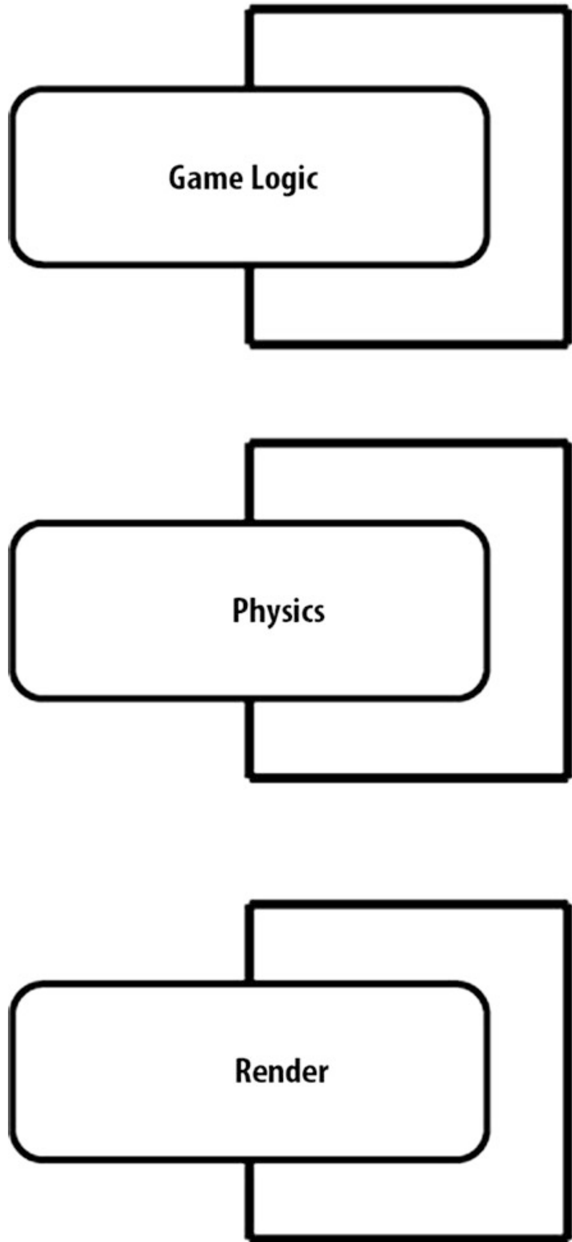
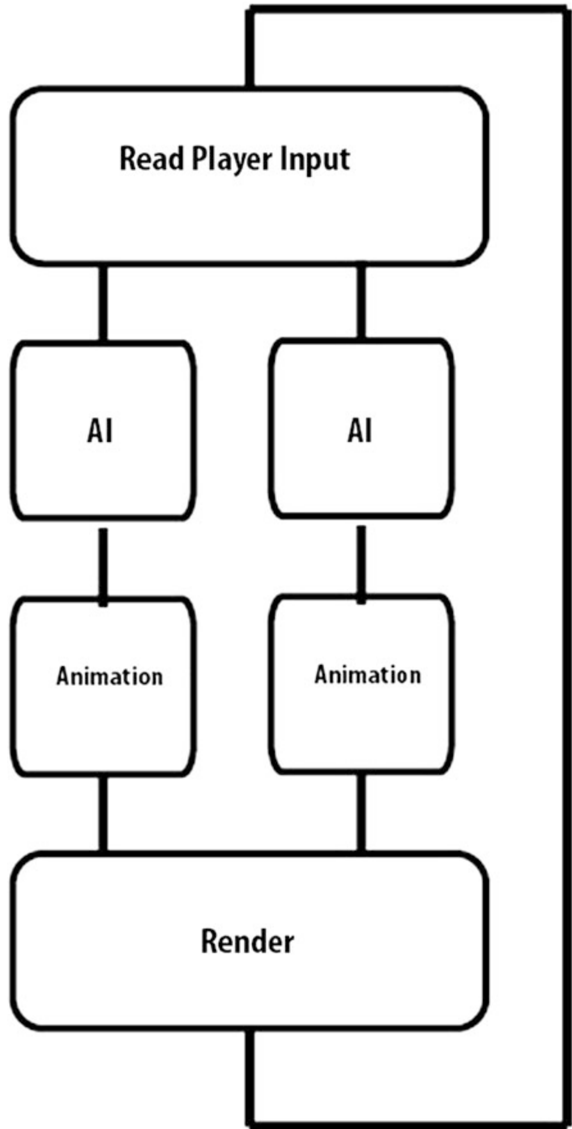


Fig. 2.6 Asynchronous function parallel model

Rhalibi et al. [15] show a different approach for real-time loops that is modeled by taking the tasks and its dependency into consideration. It divides the loop steps in three concurrent threads, creating a cyclic-dependency graph, to organize the

Fig. 2.7 Data parallel model



task ordering. In each thread, the tasks for rendering and update are divided taking into consideration their dependency.

The Data Parallel Model [12] uses a different paradigm where data are grouped in parallel sections of the application where they are processed. So, instead of using a main loop with concurrent parts that process all data, the Data Parallel Model proposes to use separate threads for sets of data (like game objects). This way, the objects run their own tasks (like AI and animation) in parallel. Figure 2.7 depicts this approach.

According to Monkkonen [12], this model scales well because it can allocate as many processing cores as they are available. Performance is limited by the amount of data processing that can run in parallel. An important issue is how to synchronize communication of objects running in different threads. The author states that the biggest drawback of this model is the need to having components designed with data parallelism in mind.

GPGPU stands for General Purpose GPU Computing. The focus on GPGPU Computing has been increasing since graphics hardware had become programmable. It is a massively parallel architecture with more powerful processing than the CPUs. GPU Computing has been theme of research on diverse areas like: image analysis [16], linear algebra [17], chemistry [18], physics simulation [19], and crowd simulation [20]. There are some works that discuss using GPU with game loops [2–6]. These works concentrate using the GPU mostly for the physics calculations, and they extend one of the game loops presented previously, i.e., multithread uncoupled model by adding a GPU stage. Figure 2.8 illustrates the Single Coupled Model with a GPGPU stage, Fig. 2.9 presents Multithread Model with a GPGPU uncoupled for the main loop, and Fig. 2.10 depicts the Multithread Uncoupled with GPGPU.

Joselli et al. [6] present an architecture for loops that is able to implement any simulation or game loop model and distribute tasks between the CPU and the GPU. We also describe a framework for loops that are able to detect the available hardware in many computers and automatically distribute tasks among the various CPU cores and also to the GPU, as Fig. 2.11 illustrates.

2.3 A Dynamic Distributed Framework Architecture

The described architecture provides a management layer that is able to analyze dynamically the hardware performance and adjust the amount of tasks to be processed by the resources, computers, CPUs, and GPUs. In order to make a correct task distribution, it is necessary to run an algorithm for the estimation. The architecture applies a scripting approach because the loop can be used in many different kinds of simulations, and for each of them it uses a different algorithm and a subset of its parameters.

The core of the proposed architecture corresponds to the Task Manager and the Hardware Check class. The Task Manager schedules tasks in threads and changes which processor handles them whenever it is necessary. The Hardware Check detects the available hardware configuration capabilities.

Additionally, with this architecture one can implement any loop model previously presented in this work. Also the heuristics presented in [4] can be adapted for this framework. An earlier version of this architecture was first presented in [6] and it is based on the concept of tasks. A task corresponds to some work that the application should execute, for instance: reading player input, rendering, and updating application objects.

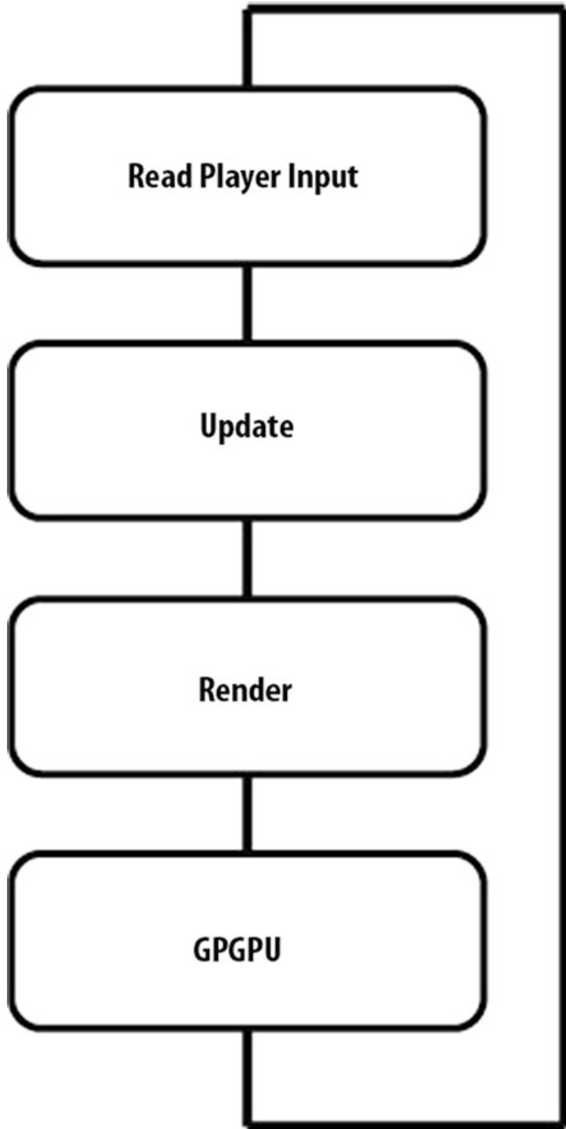


Fig. 2.8 Single coupled model with a GPGPU stage

In this architecture, a task can be anything that the application should work toward processing. However, not all tasks can be processed by all processors. Usually, the application defines three groups of tasks. The first one consists of tasks that can be modeled only for running on the CPU, like reading player input, file handling, and managing other tasks. The second group consists of tasks that run in the GPU, like the presentation of the scene. The third group can also be modeled for running on both processors and also for distributing among

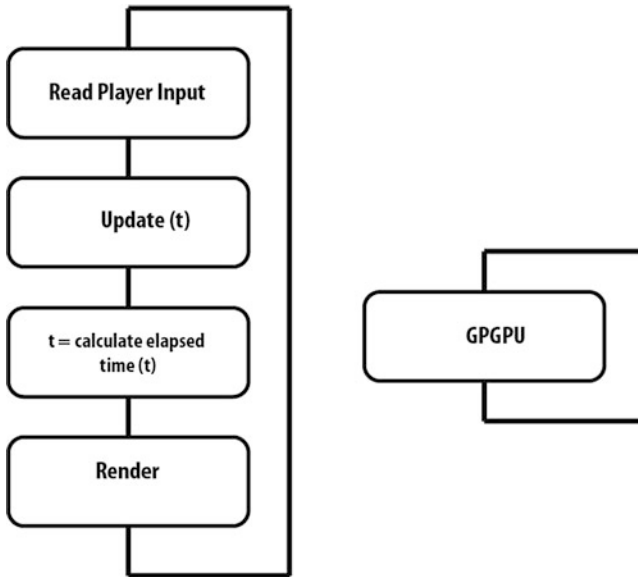


Fig. 2.9 Multithread uncoupled with GPGPU

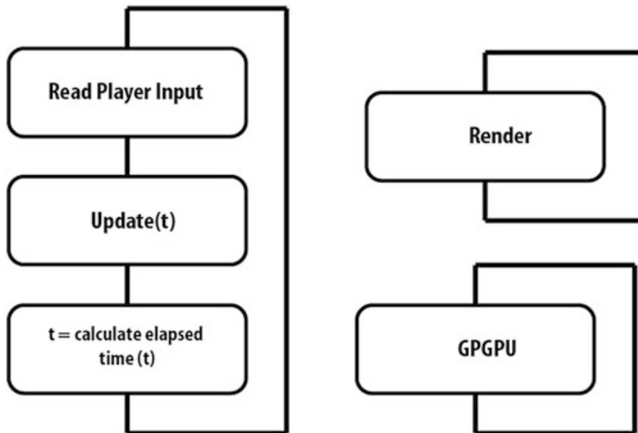


Fig. 2.10 Multithread with render uncoupled and with GPGPU stage

computers. These tasks are responsible for updating the state of some objects that belongs to the application, like AI and Physics.

The Task class is the virtual base class and has six subclasses: Input Task, Update Task, Presentation Task, Hardware Check Task, Network Check Task, and Task Manager (Fig. 2.12). The first three are also abstract classes. The fourth and fifth is a special class to check the hardware and network connection speed. The

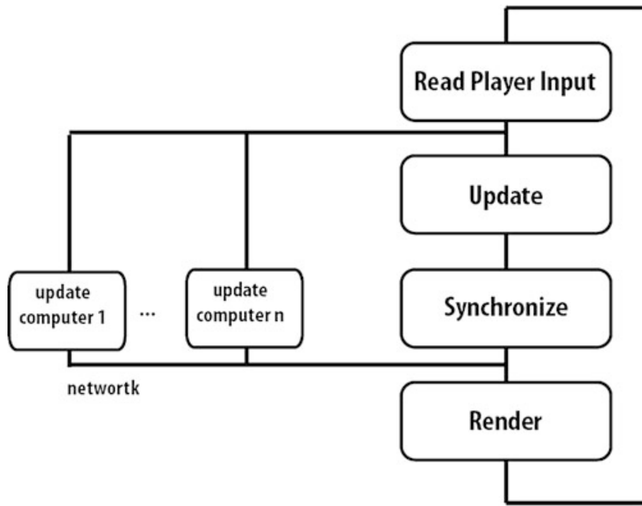


Fig. 2.11 The distributed system loop

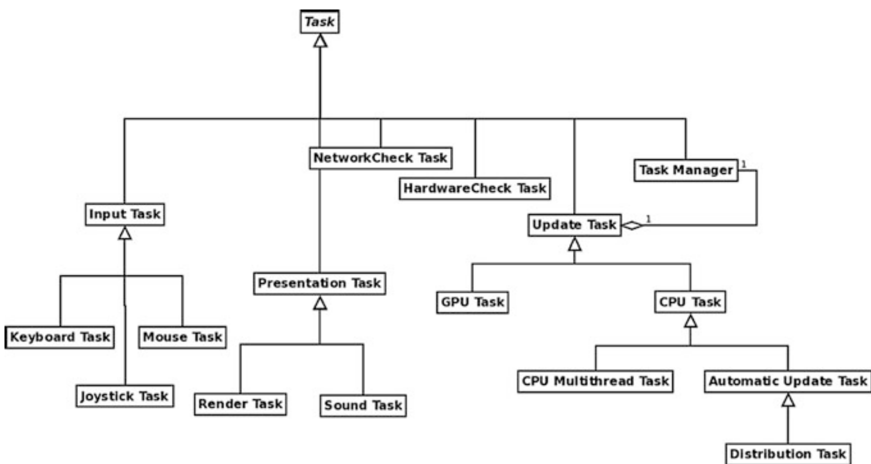


Fig. 2.12 Framework UML diagram

latter is a special class whose work consists on performing the distribution of tasks. This special class is used by the Automatic Update Task, which distributes tasks between CPU cores and GPU, and Distribution Task, which distributes tasks among computers.

The Input Task classes and subclasses handle user input-related issues. The Update Task classes and subclasses are responsible for updating the loop state. The CPU Update class should be used for tasks that run on the CPU, the GPU Update

class corresponds to tasks that run on the GPU, and the CPU Multithread task class correspond to task that can be distributed among CPUs cores. The Presentation Task and subclasses are responsible for presenting information to the user, which can be visual (Render Task) or audio (Sound Task).

The Network Check is implemented as a task that runs on the CPU. There is only one instance of this class in the application. This class checks the available computers for task processing and keeps track of the available bandwidth of the connection to each computer.

With this class the distribution task manager can know, without previous knowledge, the network connection speed to several computers. Using this information, the automatic distribution class is able to better distribute the task between the computers.

This class is always executed at the beginning of the simulation if the real-time loop model is automatic. In the case of the loop used in the simulation is a deterministic one, this class is not executed.

The Hardware Check is implemented as a task that runs on the CPU. There is only one instance of this class in the application. This class checks the available hardware and keeps track of the configuration of each computer, i.e., the number of CPU cores and GPUs (with their capabilities) available in the system.

With this class the automatic task manager can know, without previous knowledge, the available hardware for the end user computer.

This class is always executed at the beginning of the simulation if the real-time loop model is automatic. In the case of the loop used in the simulation is a deterministic one, this class is not executed.

The Task Manager (TM) is the core component of the proposed architecture. It is responsible for instancing, managing, synchronizing, and finalizing task threads. Each thread is responsible for tasks that run either on the CPU or on the GPU or on the network. In order to configure the execution of the tasks, each task has control variables described as follows:

- THREADID: the id of the thread that the task is going to use. When the TM creates a new thread, it creates a THREADID for the thread and it assigns the same id to every task that executes in that thread;
- UNIQUEID: the unique id of the task. It is used to identify the tasks;
- TASKTYPE: the task type. The following types are available: input, update, presentation, and manage;
- DEPENDENCY: a list of the tasks (ids) that this task depends on to execute.

With that information, the TM creates the task and configures how the task is going to execute. A task manager can also hold another task manager, so it can use it to manage some distinct group of tasks. An example of this case is the automatic update tasks and the distribution task.

The Task Manager acts as a server and the tasks act as its clients, as every time a task ends, it sends a message to the Task Manager. The Task manager then checks which task it should execute in the thread.

When the Task Manager uses a multithread loop model, it is necessary to apply a parallel programming in order to identify the shared and nonshared sections of the application, because they should be treated differently. The independent sections compose tasks that are processed in parallel, like the rendering task. The shared sections, like the update tasks, need to be synchronized in order to guarantee mutual-exclusive access to shared data and to preserve task execution ordering.

Although the threads run independently from each other, it is necessary to ensure the execution order of some tasks that have processing dependence. The architecture accomplishes this by using the DEPENDENCY variable list that the Task Manager checks to know the task execution ordering.

The processing dependence of shared objects needs to use a synchronization object, as applications that use many threads do. Multithread programming is a complex subject, because the tasks in the application run alternately or simultaneously, but not linearly. Hence, synchronization objects are tools for handling task dependence and execution ordering. This measure should also be carefully applied in order to avoid thread starvation and deadlocks. The TM uses semaphores as the synchronization object.

2.3.1 An Automatic and Dynamic Update Task

This is an important module that we propose. The purpose of this class is to define which processor will run the task. The class may change the task's processor during the application execution, which characterizes a dynamic distribution.

One of the major features of this new architecture is to allow dynamic and automatic task allocation between the CPU threads and GPU. In order to do that it uses the Automatic Update Task class. This task can be configured in order to be executed in five modes: one CPU thread only, multithread CPU, GPU only, in the automatic distribution between the hardware detected by Hardware Check class.

In order to execute on the multithread CPU mode, there are some requirements: a parallel CPU implementation must be provided for the CPU; for executing on the GPU mode a GPU implementation must be provided; and in order to make use of the automatic distribution all the implementations must be provided accordingly to the mode. The distribution is done by a heuristic in a script file. Also a configuration on how the heuristic is going to behave is needed, and for that a script configuration file is required. The script files can be implemented in any scripted language and in our work we developed using Lua [21].

The Automatic Update Task acts like a server and its tasks as clients. The role of the automatic update task is to execute a heuristic to automatic determine in which processor the task will be executed. The Automatic update task executes the heuristic and determines which client will execute the next task and will send a message to the chosen client, allowing it to execute. Also, every time the clients finish a task they send a message to the server to let it know it has finished. Figure 2.13 illustrates this process.

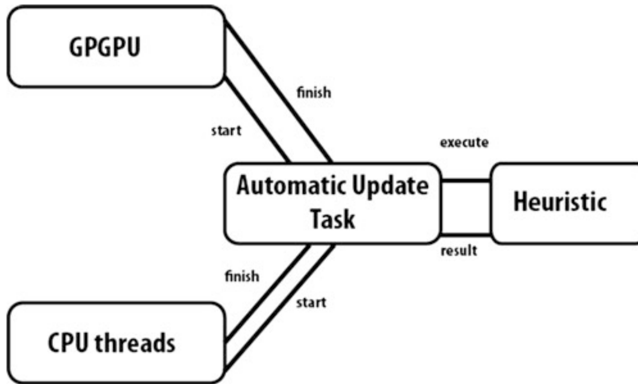


Fig. 2.13 The automatic update task class and messages

One of the main features of the proposed architecture is scheduling a task to run on another processor (CPU core to GPU or GPU to CPU core or CPU core to other CPU core) during its execution. In these cases, the task state is pushed to the task's own stack (and later restored) regardless of the processor type. For example, in time t1 the GPU processes a physics task and in time t_2 this task is scheduled to the CPU. When the task starts to run again (now in the CPU), the Task Manager reloads the task state from the tasks stack and signals it that the processor type has changed. The task priority is changed to a value of zero, which means that the task is placed on the front of the task queue. This measure is a way to guarantee that the task will keep on running. Also the Automatic Update Task can perform load balancing according to the usage rate of processors.

2.3.2 Configuration Script

The configuration script is used in order to configure how the automatic update task will execute the heuristic.

This script defines four variables:

- INITFRAMES: used in order to set how many frames are used by the heuristic to do the initial tests. These initial tests are used so the user may want the heuristic to make the initial tests differently from the normal test;
- DISCARDFRAME: used in order to discard the first DISCARDFRAME frame results, because the main thread can be loading images or models and this can affect the tests;
- LOOPFRAMES: it is used to setup how frequently the heuristic will be executed. If this value is set to -1 the heuristic will be executed only once;
- USEHARDWARE: a variable to determinate which modes will be used for the automatic update tasks;

- EXECUTEFRAMES: it is used to set how many frames are needed before the decision on changing the processor will execute the next tasks.

An example of the configuration script file can be seen in List 1.

List 1: Configuration Variables

```
INITFRAMES ⇐ 20
DISCARDFRAME ⇐ 5
LOOPFRAMES ⇐ 50
USEHARDWARE ⇐ ALLAVAILABLE
EXECUTEFRAMES ⇐ 5
```

The automatic update task begins executing after the DISCARDFRAME are executed. In the sequel, it executes INITFRAMES frames in the CPU cores and the next INITFRAMES in the GPU.

Afterward, it decides where the next LOOPFRAMES frames will be executed. If the LOOPFRAMES is greater than -1, it executes EXECUTEFRAMES frames in the CPU cores and it executes EXECUTEFRAMES frames in the GPU. Finally, it decides where the next LOOPFRAMES frames will be executed and keep repeating until the application is aborted.

The heuristic script is used in order to distribute automatically the tasks between the CPU cores and the GPU. This script defines three functions:

- reset (): reset all the variables that the script uses in order to decide which processor will execute the task. This function is called after the LOOPFRAMES frames are executed. The variable that are normally used by the heuristic are:
  - CPUTime: an array that contains the sum of all the elapsed times that the task has been processed in this CPU thread;
  - GPUTime: the sum of all the elapsed times that the task has been processed in the GPU;
  - numBodies: the number of bodies that have been processed;
  - initialBodies: the number of bodies in the beginning of the processing.
- SetVariable (elapsedTime, numberBodies, processor, thread): this function sets all the variables that the heuristic uses. This function is called after running the EXECUTEFRAMES frames in each processor. The script that defines this function can be seen on List 2.
- main (): This is the function that executes the heuristic and decides which processor will execute the task. This function is called just before the LOOPFRAMES frames is executed.

List 2: SETVARIABLE script

```
numBodies ⇐ numberBodies
if processor == CPU then
CPUTime[thread] ⇐ CPUTime[thread] + elapsedTime
else
GPUTime ⇐ GPUTime + elapsedTime
end if
```

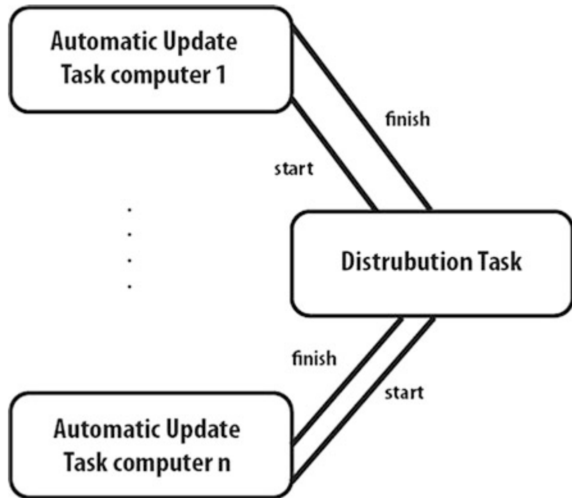


Fig. 2.14 The distribution update task class and messages

The component in the architecture enables the implementation of any real-time loop model and heuristic presented with the adaptation for distribution of tasks between cores of the same processor.

2.3.3 The Distribution Task

The purpose of this class is to define which computer will run the task. The class may change the task's computer during the application execution, which characterizes a dynamic distribution.

One of the major features of this new architecture is to allow dynamic task allocation between the computers. In order to do that it uses the Distribution Task class. This task can be configured in order to be executed in two modes: in the automatic distribution between computer with the information of the network and hardware detected by the Network Check and Hardware Check class, and in a manual mode.

The Distribution Task acts like a server and distribute the tasks between the clients. The role of the automatic update task is to execute a configuration script to determinate the execution mode of it. With that it executes the Network Check, in order to check the network between the computers, and the Hardware Check to know the configuration of each computer. The Distribution task determines which client will execute the next task and will send the task to the chosen client, allowing it to execute. Also, every time the clients finish a task they send the result to the server to let it know it has finished. Figure 2.14 illustrates this process.

Each computer have an Automatic Update Task that receives task from the Distribution Task and executes the task, when it finishes the task execution it sends the results back to the server. If a computer loses the link with the server, the server will automatic distribute its task to another client.

2.3.4 The Configuration Script

The configuration script is used in order to configure how the automatic distribution task will execute the heuristic. This script can be used to define:

- the maximum numbers of computers that will be used in the simulation;
- the computers IP addresses;
- the network mode (if it will run the tasks locally or in the network distribution mode);
- the tasks that can use the network mode;
- the minimum network connection speed between the server and the client in order to use that client machine.

2.3.5 A Summary of the Architecture Execution

First, the server computer (which displays visualization and gathers the player input) queries the network for available computers and their capacity. After that, it divides the amount of work between the computers, considering the processing power of each one. While running the game loop, the server verifies if any computer is down or if there are lost connections, in this case, it tries to redistribute the task to the other remaining computers. It also exchange messages with the computers, process its own tasks that it might have been assigned to, and then presents the results to the end user.

2.4 Conclusion

With the evolution of computer networks, distributing computation will become more in evidence, allowing more complex real-time simulations.

This work presents and summarizes the concept of the main loops present in simulations, a subject that is not very much discussed in the literature. We discuss an architecture for loops that are able to distribute tasks between computers in a network, and inside computers through CPUs, CPU cores, and GPUs. With this approach a simulator is able to use more resources available to it (local and remote), reducing its system requirements.

The framework and concept presented here can be applied to any game or real-time simulation task that can be put in a parallel mode. With the use of a distribution across the internet one could run more processing consuming simulators.

As network bandwidth increases and with the development of cloud computing, the concept of this work presents will become more and more relevant.

Acknowledgments The authors would like to acknowledge CNPq and FAPERJ for the financial support.

References

1. Glinka, F., Ploss, A., Gorlatch, S., Müller-Iden, J.: High-level development of multiserver online games. Int. J. Comput. Games Technol. **5**, 1–16 (2008)
2. Joselli, M., Zamith, M., Clua, E.W.G., Pagliosa, P., Conci, A., Montenegro, A., Valente, L.: An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. In: Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, pp. 115–120 (2008)
3. Joselli, M., Zamith, M., Valente, L., Clua, E.W.G., Montenegro, A., Conci, A., Feijó, B., Dornellas, M., Leal, R., Pozzer, C.: Automatic dynamic task distribution between CPU and GPU for real-time systems. In: IEEE Proceedings of the 11th International Conference on Computational Science and Engineering, pp. 48–55 (2008)
4. Joselli, M., Clua, E., Montenegro, A., Conci, A., Pagliosa, P.: A new physics engine with automatic process distribution between CPU–GPU. In: Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games, pp. 149–56 (2008)
5. Joselli, M., Zamith, M., Clua, E., Leal-Toledo, R., Montenegro A., Valente, L., Feijó, B., Pagliosa, P.: An architecture with automatic load balancing for real-time simulation and visualization systems. J. Comput. Interdiscip. Sci. **5**, 207–224 (2010)
6. Joselli, M., Zamith, M., Clua, E., Pagliosa, P., Conci, A., Montenegro, A., Valente, L.: (2008) An architecture with automatic load balancing and distribution for digital games. In: Proceedings of the IX Brazilian Symposium on Computer Games and Digital Entertainment, pp. 59–70
7. Valente, L., Conci, A., Feijó, B.: Real time game loop models for single-player computer games. In: Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment, pp. 89–99 (2005)
8. Dalmau, D.S.C.: (2003). Core Techniques and Algorithms in Game Programming. New Riders Publishing, US
9. Dickinson, P.: Instant replay: Building a game engine with reproducible behavior. http://www.gamasutra.com/view/feature/3057/instant_replay_building_a_game_php (2001). Accessed 2 March 2014
10. Watte, J.: Canonical game loop. http://www.mindcontrol.org/~hplus/graphics/game_loop.html (2005). Accessed 2 March 2014
11. Gabb, H., Lake, A.: Threading 3D game engine basics. http://www.gamasutra.com/view/feature/2463/threading_3d_game_engine_basics.php (2005). Accessed 2 March 2014
12. Monkkonen, V.: Multithreaded game engine architectures. http://www.gamasutra.com/view/feature/130247/multithreaded_game_engine_.php?print=1 (2006). Accessed 2 March 2014
13. Valente, L, Conci, A., Feijó, B. Real time game loop models for single-player computer games. In: Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment, pp. 89–99 (2005)

14. Microsoft: Documentation. http://msdn.microsoft.com/en-us/library/bb200104.aspx (2006). Accessed 2 March 2014
15. Rhalibi, A.E., Costa, S., England, D.: (2005) Game engineering for a multiprocessor architecture. In: DIGRA Conference
16. Kerr, A.: Campbell, D., Richards, M.: (2008) GPU VSIPL: High-performance VSIPL implementation for GPUS. High Performance Embedded Computing (2008)
17. Bolz, J., Farmer, I., Grispun, E., Schrder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multi-grid. ACM Trans. Graph. **22**(3), 917–924 (2003)
18. Ufimtsev, I.S., Martnez, T.J.: Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. J. Chem. Theory Comput. **4**(2), 222–231 (2008)
19. Nyland, L., Harris, M., Prins, J.: Fast n-body simulation with cuda. GPU Gems 3 Chapter 31, pp. 677–695 (2007)
20. Passos, E., Joselli, M., Zamith, M., Rocha, J., Montenegro, A., Clua, E., Conci, A., Feijó, B.: Supermassive crowd simulation on GPU based on emergent behavior. In: Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, pp. 81–86 (2008)
21. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua 5.1 Reference Manual. Lua.org (2006)