

# An Architecture with Automatic Load Balancing and Distribution for Digital Games

Mark Joselli  
UFF, Medialab

Marcelo Zamith  
UFF, Medialab

Luis Valente  
PUC-RIO, ICAD Games

Esteban Walter Gonzalez Clua  
UFF, Medialab

Anselmo Montenegro  
UFF, Medialab

Regina Celia P. Leal-Toledo  
UFF, IC

Bruno Feijó  
PUC-RIO, ICAD Games

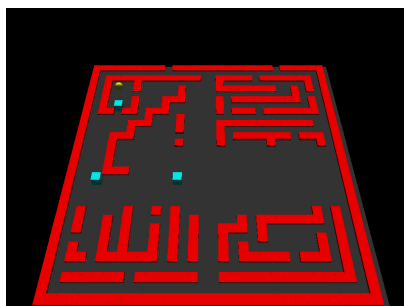


Figure 1: Screenshot of the Test Case

## Abstract

Distributed computing is being used in several fields to solve many computation intensive problems. In digital games, it is used mainly in multi-player games, where the majority of the game logic is processed in a mainframe or cluster. Single player games could also use distributed computing to process the game logic, devoting host processing to renderization, which is usually the task that digital games spend most of its processing time. By using distributed computing, games could need softer system requirements, since the game loop would be distributed. This paper presents a game loop that can be applied in both multi-player and single-player games, using automatic load balancing and distributing game logic computation among several computers.

**Keywords::** Parallel computing, Distributed computing, Digital Games, task distribution, GPGPU, real-time loop models, real-time systems, Multithread Architectures for real-time systems, Real-time applications

## Author's Contact:

mjoselli,mzamith,esteban,anselmo,leal@ic.uff.br  
lvalente,bruno@inf.puc-rio.br

## 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, like games, 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 game loops have to be carefully implemented.

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Computer Networks and others. More, computer games are also interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition in games are related to gathering data from input devices as keyboards, mice and joysticks. Data processing tasks consists on applying game rules, responding to user commands, simulating Physics and Artificial Intelligence behaviors. Data presentation tasks relate to providing feedback to the player about the current game state, usually through images and audio. In massive online games, there is also one more class for tasks:

game distribution. Game distribution is the logical partitioning of the game world among multiple servers, computation distribution management according to actual game state, and communication [Glinka et al. 2008].

Games 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 game and visual simulation performance is frames per second (FPS). The general lower acceptable bound for a game is 16 FPS. There are not 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 game loops is to better achieve an optimal FPS rate for the application.

Networking is a common feature in current computer games, and it is mainly used for providing multi-player game sessions and updating game data. The prominent usage of networking in games has given birth to a new class of games named MMOGs (Massively Multi-player Online Games).

The architecture, this paper presents, follows a similar concept as cloud computing, where computers across the Internet shares resources, software and information, where the players' computer can use other resources available on the network, to help it in processing the game. By using this approach, a computer with less computing power could join the game session, by relaying the effort to process the game to the network cloud.

A similar approach of this work is the Onlive service ([www.onlive.com](http://www.onlive.com)), which runs all the game loop tasks inside a computer cluster, with the end user being only responsible for processing the video renderization and gathering the users' input. This approach consume less power than the required for processing of the game renderization and controlling input, but it needs a high connection speed and is subject to high latency.

This work extends the work [Joselli et al. 2010], which presented a framework for game loops that use automatic task distribution between CPU cores and GPU, for single or multi-threaded loops. This work adds the following concepts:

- distribution of tasks related to the game loop, between computers;
- Load balancing of the tasks.

This paper is organized as follows: Section 2 presents a set of real-time game loop model concepts found in the literature. Section

3 presents the framework architecture. Section 4 presents the test case, a parallel implementation of the Pac-man game, along with the test results and discussion. Finally, Section 6 presents the conclusions of this work.

## 2 Real-Time Loop Models

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.

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, and motion sensors. 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, simulating the Artificial Intelligence, and related tasks. The presentation refers to providing feedback to the user about the current simulation state, through images and audio.

A real-time system like games and some visual simulations, 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 and games, it is not easy to find academic works specifically devoted to this subject. The works by Valente et al. [Valente et al. 2005], Dalmau [Dalmau 2003], Dickinson [Dickinson 2001], Watte [Watte 2005], Gabb and Lake [Gabb and Lake 2005], and Mönkkönen [Mönkkönen 2006] are among the few ones.

The simplest real time loop models are the coupled ones. The Simple Coupled Model [Valente et al. 2005] is perhaps the most straightforward approach to modeling real-time loops. It consists of sequentially arranging the tasks in a main loop. Figure 2 depicts this model.

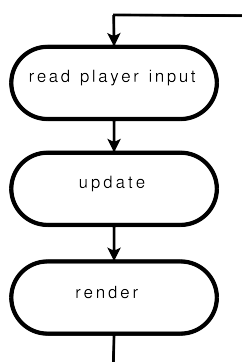


**Figure 2:** *Simple Coupled Model*

The uncoupled models separate the rendering and update stages, so they can run independently, in theory. These models consider single-thread [Valente et al. 2005; Dickinson 2001] and multithread designs [Valente et al. 2005; Gabb and Lake 2005; Mönkkönen 2006]. The Multithread Uncoupled Model [Valente et al. 2005] and the Single-thread Uncoupled Model [Valente et al. 2005] try to bring determinism to the game execution by feeding the update stage with a time parameter. Figures 3 and 4 illustrate these models, respectively.

By using these models, the application has a chance to adjust its execution with time, so the game can run the same way in different machines. More powerful machines will be able to run the game more smoothly, while less powerful ones will still be able to provide some experience to the user.

Although these are working solutions, time measuring may vary greatly in different machines due to many reasons (such as process load), making it difficult to reproduce it faithfully. For example, some simulations may require a scene replay feature [Dickinson

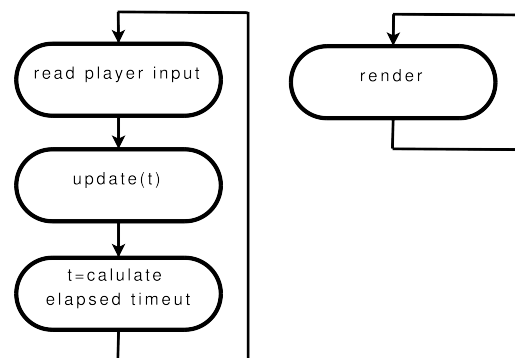


**Figure 3:** *Multithread Uncoupled Model*

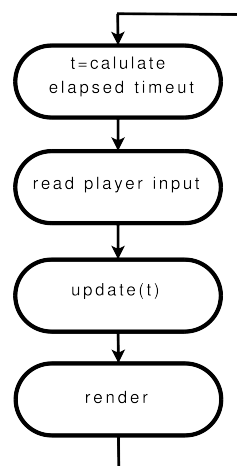


**Figure 4:** *Single-thread Uncoupled Model*

2001], 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 [Dickinson 2001] 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.

Hence, the models proposed in [Valente et al. 2005; Dickinson 2001; Microsoft 2006] try to address these issues. The Fixed-frequency Uncoupled Model outlined in [Valente et al. 2005] features another update stage that runs at a fixed frequency, besides the time-based one. The work by Dalmau [Dalmau 2003] present a similar model, although not naming it explicitly. These works describe the model using a single-thread approach. Figure 5 illustrates the Fixed-frequency Uncoupled Model.

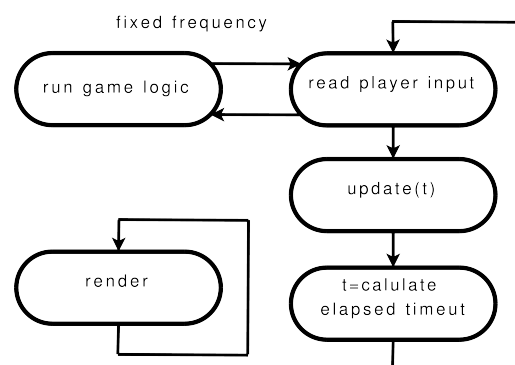


**Figure 5:** *Fixed-frequency Uncoupled Model*

The model described in [Dickinson 2001] 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 [Microsoft 2006]. 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.

Nowadays, computers and new video game consoles (such as the Xbox 360 and the Playstation 3) feature multi-core processors. For this reason, real-time loops that take advantage of these resources are likely to become important in the near future. Therefore, making the tasks parallel in multiple threads is a natural step.

However, dealing with concurrent programming introduces another set of problems, such as data sharing, data synchronization, and deadlocks. Also, as Gabb and Lake [Gabb and Lake 2005] states, that not all tasks can be fully parallelized due to dependencies among them. As examples, in a game, characters cannot move until the game logic is computed, and rendering cannot be performed until the game state is updated. Hence, serial tasks represent a bottleneck to parallelizing simulation computation.

The work [Mönkkönen 2006] presents models regarding multi-thread 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 [Mönkkönen 2006] proposes to allocate a thread to all tasks that are (theoretically) independent of each other. For example, performing Physics simulation while calculating animation. Figure 6 illustrates this model.

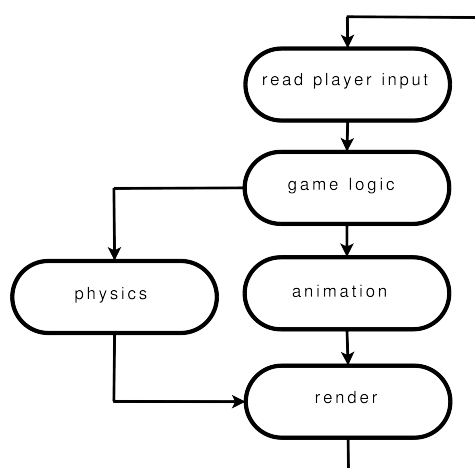


**Figure 6:** *Synchronous Function Parallel Model*

Mönkkönen 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 [Mönkkönen 2006] is the formalization of the idea found in [Gabb and Lake 2005]. This model does not present a main loop. Figure 7 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 rendering 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 [Gabb and Lake 2005].

Rhalibi et al. [Rhalibi et al. 2005] shows 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 task ordering. In each thread, the tasks for rendering and update are divided taking into consideration their dependency.

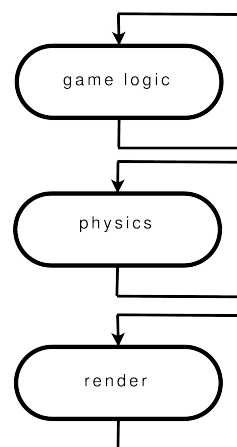


**Figure 7:** *Asynchronous Function Parallel Model*

The Data Parallel Model [Mönkkönen 2006] 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 8 depicts this approach.

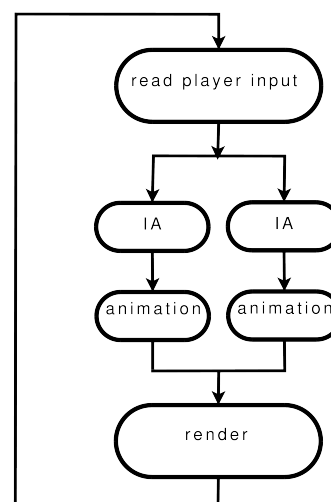


**Figure 8:** *Data Parallel Model*

According to the author [Mönkkönen 2006], 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.

The focus on GPGPU has been increasing since graphics hardware had become programmable. It is a massively parallel architecture with more powerful processing than the CPUs. GPGPU has been theme of research on diverse areas like: image analysis [Kerr et al. 2008], linear algebra [Bolz et al. 2003], chemistry [Ufimtsev and Martinez 2008], physics simulation [Nyland et al. 2007], and crowd simulation [Passos et al. 2008]. There are some works that discuss using GPGPU with game loops [Joselli et al. 2008b; Joselli et al. 2008c; Joselli et al. 2008a; Zamith et al. 2007; Zamith et al. 2008]. These works concentrate using the GPU mostly for the physics calculations, and they extend one of the game loops presented previously, i.e., multi-thread uncoupled model by adding a GPGPU stage. Figure 9 illustrates the Single Coupled Model with an GPGPU stage, Figure 10 presents Multithread Model with a GPGPU uncoupled for the main loop, and Figure 11 depicts the Multithread Uncoupled with GPGPU.

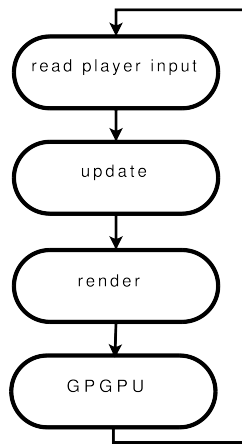


**Figure 9:** *Single Coupled Model with an GPGPU stage*

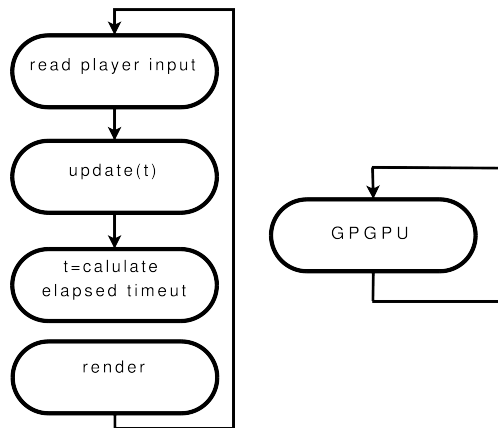


**Figure 10:** *Multithread Uncoupled with GPGPU*

Joselli et al [Joselli et al. 2010] presents an architecture for game loops is able to implement any game loop model and distribute tasks between the CPU and the GPU. This work extends the work by Joselli et al. by proposing a framework for game loops that is able to detect the available hardware in many computers and automatically distribute tasks among the various CPU cores and also to the GPU, as figure 12 illustrates.

### 3 The Framework Architecture

The aim of the proposed architecture is to provide 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, and in the current architecture, a script is responsible for this. The architecture applies the scripting approach because the loop can be used in many simula-

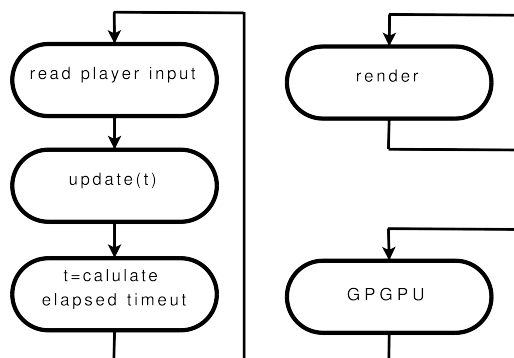


**Figure 11:** *Multithread Uncoupled with GPGPU*

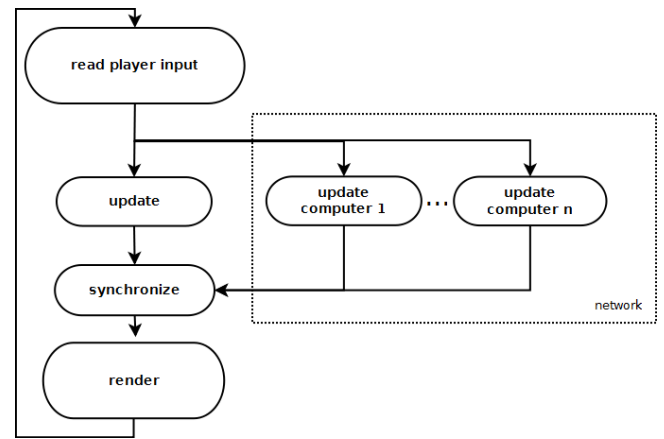


**Figure 12:** *The distributed game loop*

tions, 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 present in this work. Also the heuristics presented in [Joselli et al. 2008a] can be adapted for this framework. An earlier version of this architecture was first presented in [Joselli et al. 2010] 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.

In the proposed architecture, a task can be anything that the application should work towards 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 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. The first three are also abstract classes. The fourth and fifth is a special class to check the hardware and network connection speed. The latter is a special class whose work consist 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).

#### 3.1 The Network Check Class

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.

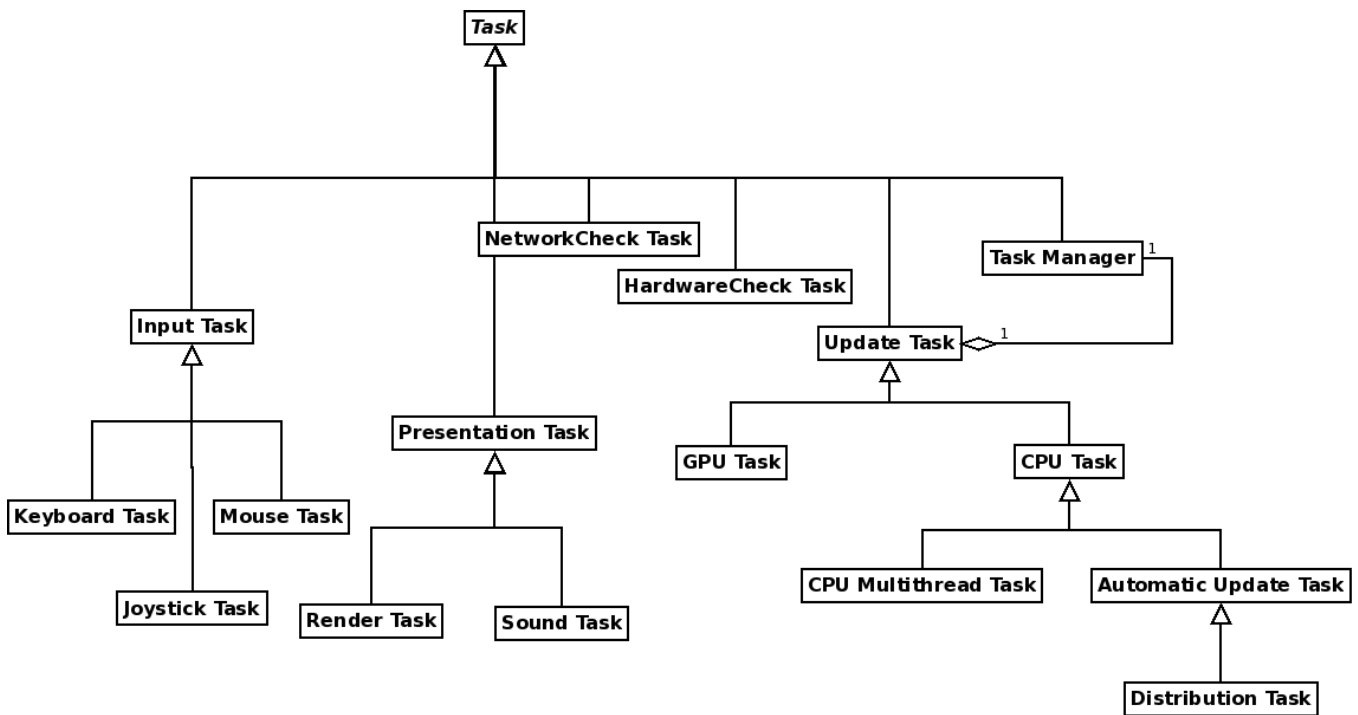


Figure 13: Framework UML Diagram

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.

### 3.2 The Hardware Check Class

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.

### 3.3 The Task Manager

The Task Manager (TM) is the core component of the proposed architecture. It is responsible for instantiating, 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 multi-thread loop model, it is necessary to apply a parallel programming in order to identify the shared and non-shared 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. Multi thread 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.

### 3.4 The Automatic Update Task

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, multi-thread CPU, GPU only, in the automatic distribution between the hardware detected by Hardware Check class.

In order to execute on the multi-thread 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 presented in Subsection 3.5.1. The scripts files are implemented in Lua [Ierusalimsky et al. 2006] (Subsection 3.4.2).

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 14 illustrate this process.

One of the major 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 tasks own stack (and later restored) regardless of the processor type. For example, in time  $t_1$  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.

### 3.4.1 The 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 discards 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 set up 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 scrip file can be seen in algorithm 1.

---

#### Algorithm 1 Configuration Script

---

```
INITFRAMES  $\Leftarrow$  20
DISCARDFRAME  $\Leftarrow$  5
LOOPFRAMES  $\Leftarrow$  50
USEHARDWARE  $\Leftarrow$  ALLAVAILABLE
EXECUTEFRAMES  $\Leftarrow$  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.

Afterwards, 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.

### 3.4.2 The Heuristic Script

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;
  - **GPUPTime**: 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 algorithm 2.

---

#### Algorithm 2 SetVariable Script

---

```
numBodies  $\Leftarrow$  numberBodies
if processor == CPU then
  CPUTime[thread]  $\Leftarrow$  CPUTime[thread] + elapsedTime
else
  GPUPTime  $\Leftarrow$  GPUPTime + elapsedTime
end if
```

---

- **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 are executed.

The component in the architecture enables the implementation of any real-time loop model and heuristic presented in section 5 with the adaptation for distribution of tasks between cores of the same processor.

## 3.5 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 determine 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

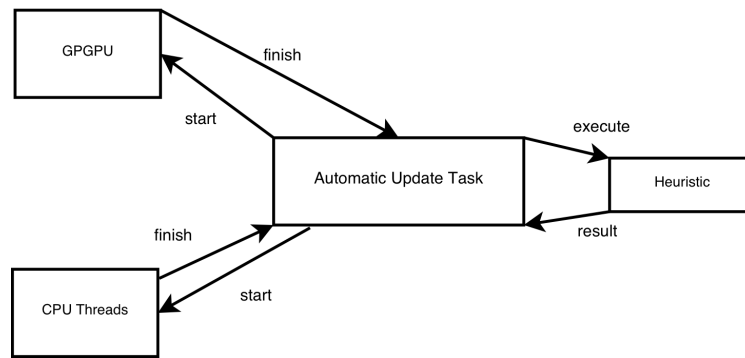


Figure 14: The Automatic Update Task class and messages

task they send the result to the server to let it know it has finished. Figure 15 illustrate this process.

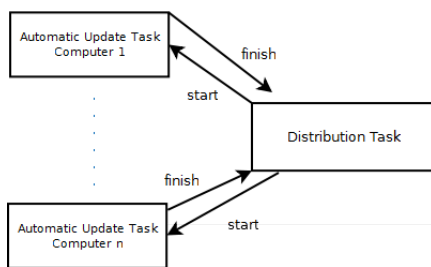


Figure 15: The Distribution Update Task class and messages

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 lose the link with the server, the server will automatic distribute its task to another client.

### 3.5.1 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.

## 3.6 The Architecture Execution

This subsection is dedicated to illustrate the execution of the architecture, so the reader can better understand it. Figure 16 illustrates the process.

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 re-distribute 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.

## 4 Test Case

The case study implements a simple distributed version of path-finding through a Pac-man game. Indeed, the goal of games such as Pac-man are basically finding out the best path for a target, i.e., the Pac-man character. Although simple games such as Pac-man do not require a intense processing, this approach is adopted in order to validate the propose game loop. Figure 1 illustrate a screenshot of the game.

The game was developed using OpenGL in order to realize the visualization, whereas MPI (Message Protocol Interface) was adopted to exchange messages among network nodes. When the application is initialized, it searches for other peers in the network, defining the number of the number of network nodes, i.e., the numbers of computers in the network. The node that starts the application is deemed as the server node, whereas the others nodes are considered as clients. With this arrangement, in each time step the game loop of the server application sends the Pac-man position to the clients, and afterwards receives the positions of the ghosts from them. The server runs all game loop stages (input player, update and visualization), whereas the clients run only the update stage. When the game ends, the application broadcasts a message to all peers informing about this event.

The test network is composed by three computers. One of them use an Intel Quad Core with 2.4GHz, 4GB and nVidia GeForce 8400 GS graphic card. The other two computers use an Intel Core 2 dual with 2.4 GHz, 2GB of memory and nVidia GeForce 8800 GTS graphic card. All computers use the Ubuntu 9.10 operation system (64bits) and OpenMPI. The network bandwidth is 1.0 Gbits. The server computer is the one with the Quad Core processor, while the others act as clients.

In order to measure the work of each node, the tests created ghosts according to multiples of the number of processing cores. Therefore, the tests used three and six ghosts. This corresponds to four and seven processes respectively. It is a process for each ghost plus one process for the client.

Whenever the number of processes is greater than the number of nodes, a node is allocated more than one process. For instance, using one node and six ghosts, all processes run in the same node. In this case, there's no communication overhead.

The test case (the Pac-man game) distributes the ghosts' AI among network nodes, with a total of six test scenarios. The first two scenarios use only one node, with three ghosts for scenario 1 and six ghosts for scenario 2. The next two scenarios use two nodes, with three and six ghosts, respectively. The last two scenarios use three nodes, with three ghosts in scenario 5 and six in scenario 6.

When the number of ghosts is greater then the number of nodes, each node processes more than one ghost. The simulation computes the average elapsed time for 1.010.000 frames, where the firsts 10.000 are discarded in order to stabilize the system.

In Table 1, the Column *Nodes/Ghost* represents the number of nodes and number of ghosts respectively. The *FPS* Column illustrates the amount of frames per second. The *Elapsed time of the*

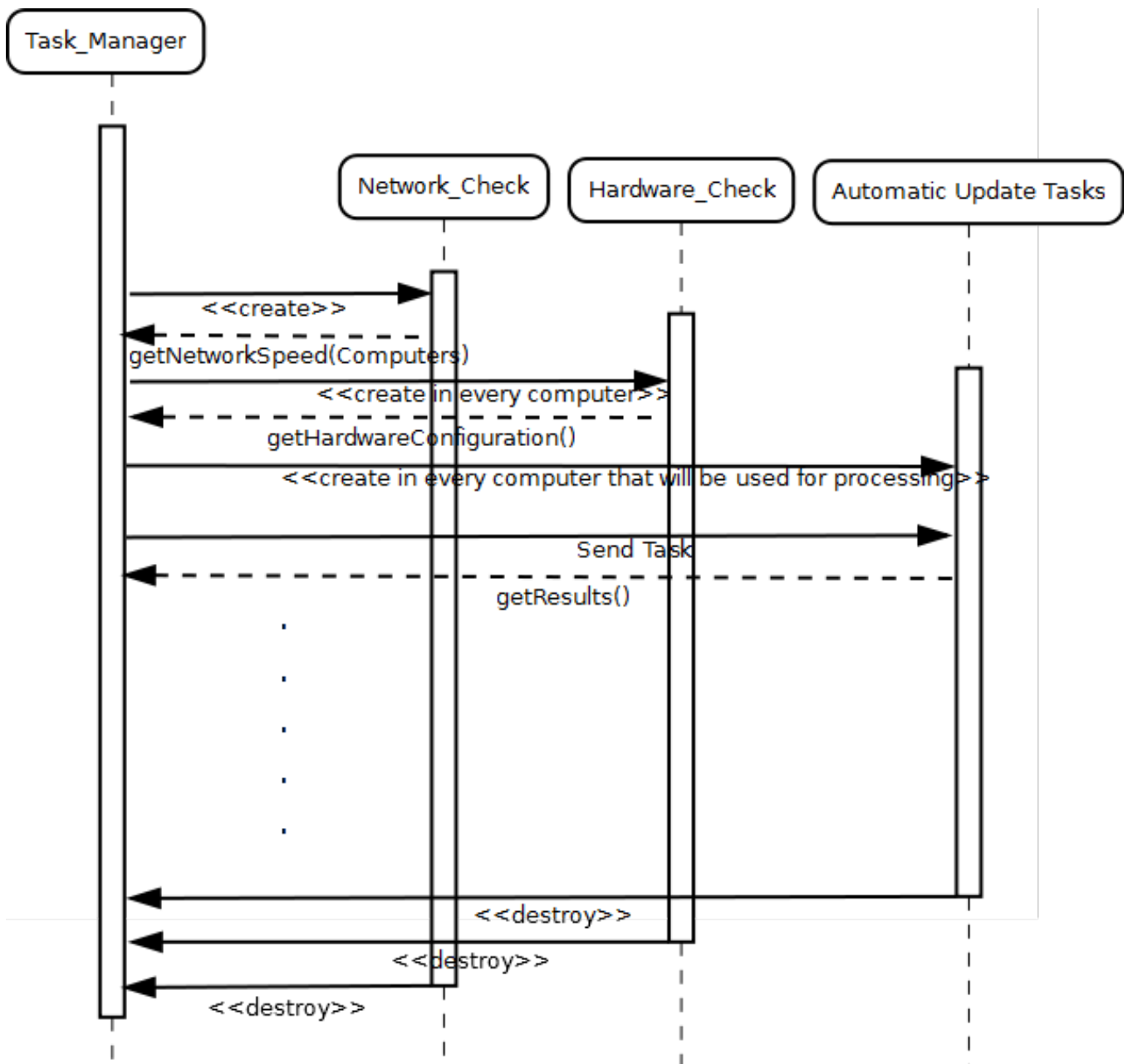


Figure 16: The Illustration of the Execution Process

entire game loop Column contains the elapsed time in processing all steps of update stage, measured in milliseconds. Finally, the *Elapsed time communication* Column represents the elapsed time of the network communication, measured in milliseconds and in the Column *Percent spent in communication* the amount of percentage of the game loop that is spent with communication. The results obtained indicate that communication represents less than 1% of all processing. In other words, the communication is not the "bottleneck". Even if we increase the number network nodes (and consequently, increasing message exchanging), the total elapsed time decreased. This means the distributed path-finding increased the game performance.

Figure 17 illustrates a chart with ghosts per nodes and the corresponding frames per second measurement. Figure 18 presents a chart with nodes per ghosts and the corresponding elapsed times, considering processing and network communication.

These results indicate that with more machines (nodes), the simulation is faster (a speed-up of 1,38) when processing 6 ghosts using 3 machines, compared to the 1-machine case. Also, these results indicate that the time spent in communications among computers do not influence application performance.

## 5 Conclusion

With the evolution of computer networks, distributing computation will become more in evidence, even for real-time simulations like games.

This work discussed the concept of game loops, a subject that is not very much discussed in the literature. Our contribution lies on extending a previous work, by providing an architecture for game loops that is able to distribute tasks between computers in a network, and inside computers through CPUs, CPU cores and GPUs. With this approach a game 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. And with the use of a distribution across the internet one could run more processing consuming games in with less minimum requirements for it.

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

| Nodes/Ghosts | FPS    | Elapsed time of the entire game loop | Elapsed time communication | Percent spent in communication |
|--------------|--------|--------------------------------------|----------------------------|--------------------------------|
| 3/6          | 310.57 | 3.20393                              | 0.01600                    | 0.00497                        |
| 3/3          | 337.36 | 2.95722                              | 0.00700                    | 0.00236                        |
| 2/6          | 261.94 | 3.80467                              | 0.01300                    | 0.00341                        |
| 2/3          | 285.93 | 3.49138                              | 0.00600                    | 0.00172                        |
| 1/6          | 228.65 | 4.37144                              | 0.00200                    | 0.00046                        |
| 1/3          | 244.05 | 4.09657                              | 0.00100                    | 0.00024                        |

**Table 1:** Elapsed time expent with processing and communication.

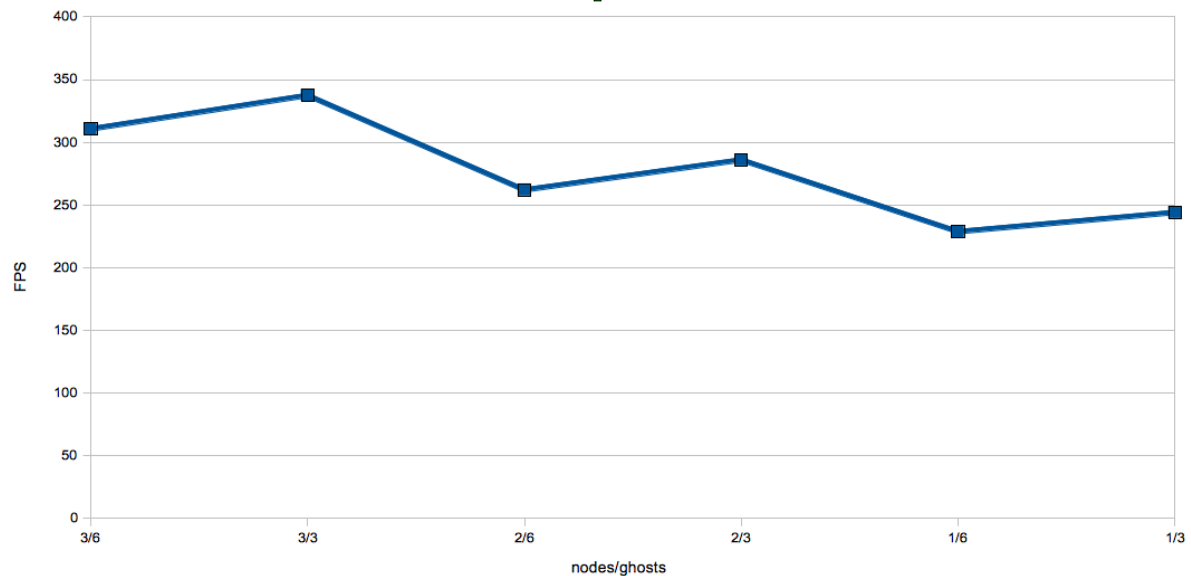


**Figure 17:** Elapsed time of the simulation in FPS.

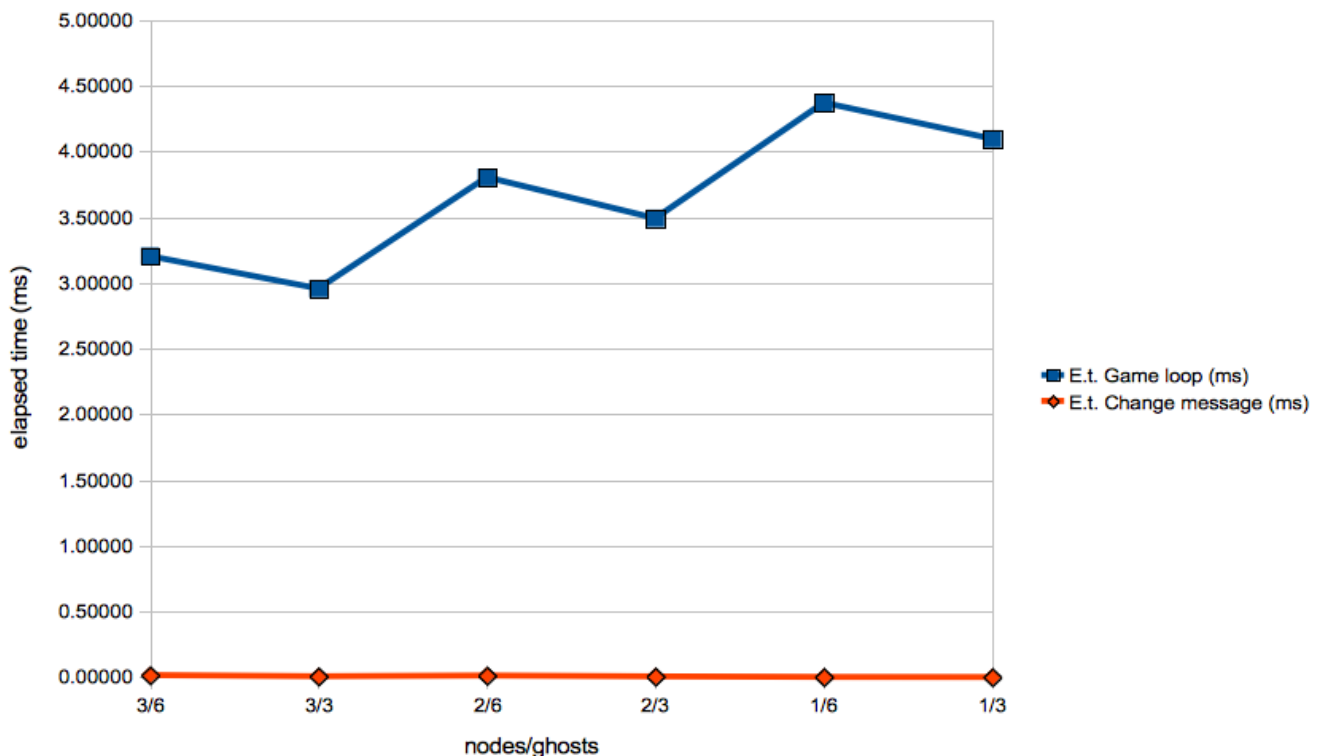


**Figure 18:** Elapsed time of the processing and the communication of the simulation.

## References

- BOLZ, J., FARMER, I., GRISPUN, E., AND SCHRDER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multi-grid. *ACM Transactions on Graphics* 22, 3, 917–924.
- DALMAU, D. S. C. 2003. *Core Techniques and Algorithms in Game Programming*. New Riders Publishing.
- DICKINSON, P., 2001. Instant replay: Building a game engine with reproducible behavior. Available at <http://www.gamasutra.com/features/>

- 20010713/dickinson.01.htm/ .
- GABB, H., AND LAKE, A., 2005. Threading 3d game engine basics. Available at <http://www.gamasutra.com/features/20051117/gabb.01.shtml/> .
- GLINKA, F., PLOSS, A., GORLATCH, S., AND MÜLLER-IDEN, J. 2008. High-level development of multiserver online games. *International Journal of Computer Games Technology* 2008, 5, 1–16.
- IERUSALIMSKY, R., DE FIGUEIREDO, L. H., AND CELES, W. 2006. *Lua 5.1 Reference Manual*. Lua.org.
- JOSELLI, M., ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2008. An adaptive game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 115–120.
- JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., FEIJ, B., DORNELLAS, M., LEAL, R., AND POZZER, C. 2008. Automatic dynamic task distribution between cpu and gpu for real-time systems. *IEEE Proceedings of the 11th International Conference on Computational Science and Engineering*, 48–55.
- JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 149–156.
- JOSELLI, M., ZAMITH, M., CLUA, E., LEAL-TOLEDO, R., MONTENEGRO, A., VALENTE, L., FEIJO, B., AND PAGLIOSA, P. 2010. An architecture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, 207–224.
- KERR, A., CAMPBELL, D., AND RICHARDS, M. 2008. Gpu vsipl: High-performance vsipl implementation for gpus. In *High Performance Embedded Computing*.
- MICROSOFT, 2006. Xna 1.0 documentation. Available at [help://MS.VSEExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/AppModel\\_GameLoop.htm](http://help://MS.VSEExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/AppModel_GameLoop.htm) .
- MÖNKKÖNEN, V., 2006. Multithreaded game engine architectures. Available at <http://www.gamasutra.com/features/20060906/monkkonen.01.shtml> .
- NYLAND, L., HARRIS, M., AND PRINS, J. 2007. Fast n-body simulation with cuda. *GPU Gems 3 Chapter 31*, 677–695.
- PASSOS, E., JOSELLI, M., ZAMITH, M., ROCHA, J., MONTENEGRO, A., CLUA, E., CONCI, A., AND FEIJÓ, B. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 81–86.
- RHALIBI, A. E., COSTA, S., AND ENGLAND, D. 2005. Game engineering for a multiprocessor architecture. In *DIGRA Conf*.
- UFIMTSEV, I. S., AND MARTNEZ, T. J. 2008. Quantum chemistry on graphical processing units. I. strategies for two-electron integral evaluation. *Journal Chemistry Theory Computation* 4 (2), 222 – 231.
- VALENTE, L., CONCI, A., AND FEIJÓ, B. 2005. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 89–99.
- WATTE, J., 2005. Canonical game loop. Available at [www.mindcontrol.org/~hplus/graphics/game\\_loop.html/](http://www.mindcontrol.org/~hplus/graphics/game_loop.html/) .
- ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2007. The gpu used as a math co-processor in real time applications. *Proceedings of the VI Brazilian Symposium on Computer Games and Digital Entertainment*, 37–43.
- ZAMITH, M. P. M., CLUA, E. W. G., CONCI, A., MONTENEGRO, A., LEAL-TOLEDO, R. C. P., PAGLIOSA, P. A., VALENTE, L., AND FEIJÓ, B. 2008. A game loop architecture for the gpu used as a math coprocessor in real-time applications. *Comput. Entertain.* 6, 3, 1–19.