Journal of Computational Interdisciplinary Sciences (2010) 1(3): 207-224

ISSN 1983-8409
http://epacis.org

An architecture with automatic load balancing for real-time simulation and visualization systems

Mark Joselli[1], Marcelo Zamith[1], Esteban Clua[1], Regina Leal-Toledo[1], Anselmo Montenegro[1], Luis Valente[2], Bruno Feijó[2] and Paulo Pagliosa[3]

Manuscript received on September 15, 2009 / accepted on March 01, 2010

ABSTRACT

Nowadays, multithread hardware architectures like multi-core CPUs and GPUs found on PCs and game consoles (as Microsoft Xbox 360 and Sony Playstation 3) are a trend. Hence, real-time simulation and visualization systems, such as scientific visualization, games and virtual reality environments, will not get the best performance on such architectures running sequentially in a single-thread loop. For this reason, multithread real-time loop models that take advantage of such architectures are gaining importance. This paper presents a survey on loop models for games and real-time systems. Also it discusses the usage of simple loops with single-thread architecture and multithread loop architectures in scientific simulations and visualization systems. Furthermore, this paper presents a new architecture for real-time loops that can detect and analyze the user hardware in order to adapt itself to a specific loop model, achieving the best performance for a specific hardware and application.

Keywords: parallel computing, task distribution, GPGPU, real-time loop models, real-time systems, multithread architectures for real-time systems, real-time applications.

[1]Instituto de Computação, Universidade Federal Fluminense, Rua Passo da Pátria, 156, Bloco E, 3° andar, São Domingos, 24210-240 Niterói, RJ, Brazil.

[2]VisionLab, Pontifícia Universidade Católica do Rio de Janeiro, Rua Marques de São Vicente, 225, Gávea, 22453-900 Rio de Janeiro, RJ, Brazil.

[3]Faculdade de Computação, Universidade Federal de Mato Grosso do Sul, Campus de Campo Grande, Unidade II, sala 3, 79070-900 Campo Grande, MS, Brazil.

E-mails: mjoselli@ic.uff.br, mzamith@ic.uff.br, esteban@ic.uff.br, leal@ic.uff.br, anselmo@ic.uff.br, lvalente@inf.puc-rio.br, bfeijo@inf.puc-rio.br, pagliosa@facom.ufms.br

1 INTRODUCTION

Multithread architectures on PC are getting more and more common with the development of multi-core processors and the new GPU parallel architectures that can also be used for generic processing. Also top of the line video games like the Microsoft Xbox 360 and the Sony Playstation 3 feature multi-core processors. Hence, real-time architectures have to parallelize and distribute their tasks between the available processors, using concepts from distributed and parallel systems in order to fully take advantage of the hardware. This work utilizes some of these concepts, like task distribution and load balancing, adapting them to the loop architectures of real-time simulations.

Games and simulations are interactive real-time systems and, like most multimedia applications, they have time constraints to execute all of their processes and to present the results to the end user. If a simulation does not fulfill this requirement, it will lose its interactivity and consequently it will fail. A common parameter for measuring a game and simulations performance is frames per second (FPS). The lower acceptable bound for a game is 16 FPS. There are not higher bounds for a game FPS, but in PCs when the refresh rate of the monitor is less than the refresh of the application, disposals of the rendered frame may occur. In order to achieve the best FPS in the application, real-time systems loops are designed and developed.

The simulation loop is the structure that determines the order in which each task of the application is executed during the loop. The loops are mainly divided in three stages: data acquisition, which gets the data from user's input; data processing, where the application logic is processed and the simulation state is updated; and data presentation, where the results are presented to the end user through images and audio. There are several works that deal with real-time loops in order to achieve better results but normally they are very restricted to the designed hardware. This work presents a framework capable of executing game loops that can be dynamically adapted to the user hardware in order to take the best performance from it.

When a task is divided in threads, in the distribution of the computation between the threads sometimes is not ideal making threads wait for each other. This is a common problem in a distributed system and, to solve this, an automatic work distribution among the threads is needed. The proposed framework implements this load balancing using a heuristic to solve this problem.

This paper is organized as follows. Section 2 presents real-time loop models. Section 3 presents GPGPU concepts. Section 4 presents GPUs real-time loop models. Section 5 presents distributed real-time loop models and also heuristics for distribution of the tasks of a real-time loop. Section 6 presents a framework to implement any real-time loop model, with load balancing and adaptation of the loop to the end user hardware configuration. Section 7 presents the test case and results. Finally, Section 8 presents the conclusions.

2 REAL-TIME LOOP MODELS

Real-time loops are the underlying structures games and real-time simulations are built upon. These loops are regarded as real-time because games and simulations (and similar kinds of multimedia applications) have time constraints to run the tasks that rely on them. This means that if those tasks do not run fast enough, the experience that the application must provide will be compromised.

The tasks that a computer simulation should execute can be broken down into three general stages: 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 stage 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 others tasks to update the simulation state. The presentation refers to providing feedback to the user about the current simulation state, through images and audio.

As listed previously, there are many tasks that a real-time system must run. A real-time system provides the illusion that everything is happening at once. Since simulations and games are interactive applications, if they are unable to perform its work on time, the user experience will not be acceptable. This issue characterizes real-time computer simulations and games as heavy real-time applications.

Although 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. [27], Dalmau [3], Dickinson [4], Watte [28], Gabb and Lake [5], and Monkkonen [17] are among the few ones.

The simplest real-time loop models are the coupled ones. The Simple Coupled Model [27] is perhaps the most straightforward approach to modeling real-time loops. It consists of sequentially arranging the tasks in a main loop. Figure 1 depicts this model.

This first model runs as fast as the machine is able to, making it very unpredictable when it is used in different machine configurations. The Synchronized Coupled Model [27] tries to repair

this issue by delaying the main loop to meet a predefined running frequency. The purpose is to bring uniformity to the application execution. However, this strategy poses an artificial limit to the application, possibly wasting processing power in order to meet that requirement. Figure 2 illustrates this model.

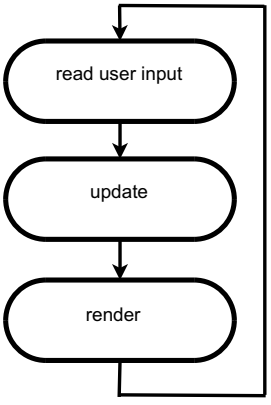

Figure 1 – Simple Coupled Model.

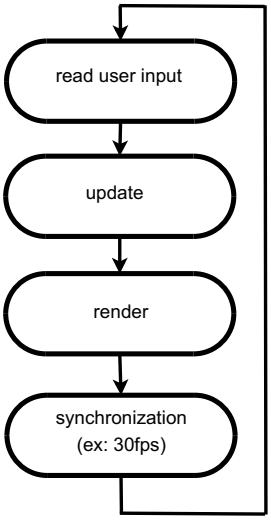

Figure 2 – Synchronized Coupled Model.

The uncoupled models separate the rendering and update stages, so they can run independently, in theory. These models consider single-thread [27, 4] and multithread designs [27, 5, 17]. A naive approach to a real-time uncoupled models is to use one thread for rendering and another for the update tasks. This approach exposes the same unpredictable behavior of the Simple Coupled Model. The Multithread Uncoupled Model [27] and the Single-thread Uncoupled Model [27] 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.

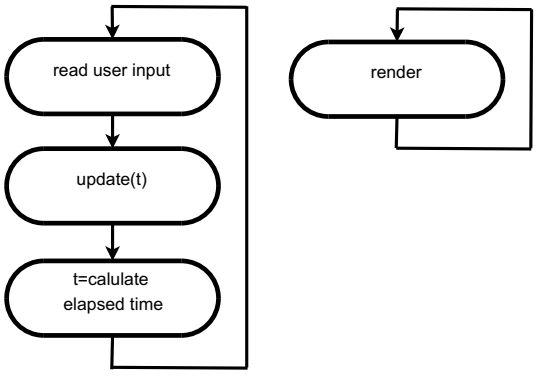

Figure 3 – Multithread Uncoupled Model.

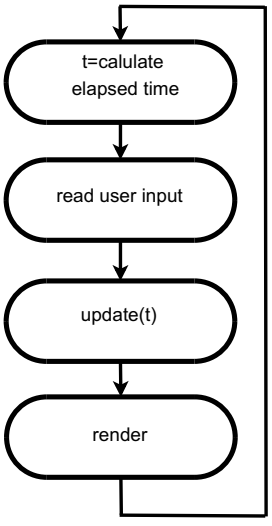

Figure 4 – Single-thread Uncoupled Model.

By using these models, the application has a chance to adjust its execution time and run the same way in different machines. More powerful machines will be able to run the application 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 do reproduce it faithfully. For example, some simulations may require a scene replay feature [4], 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 [4] 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 logic, may not yield better results. Hence, the models proposed in [27, 4, 16] try to address these issues. The Fixed-frequency Uncoupled Model outlined in [27] features another update stage that runs at a fixed frequency, besides the time-based one. The work by

Dalmau [3] present a similar model, although not naming it explicitly. These works describe the model using a single-thread approach. Figure 5 illustrates it.

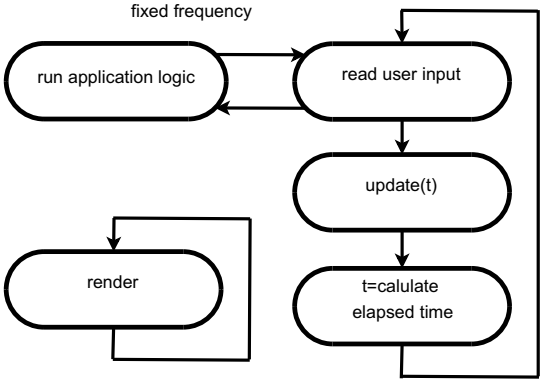

Figure 5 – Fixed-frequency Uncoupled Model.

The model described in [4] 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 [16]. 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 [5] states, not all of 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 [17] 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 [17] 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.

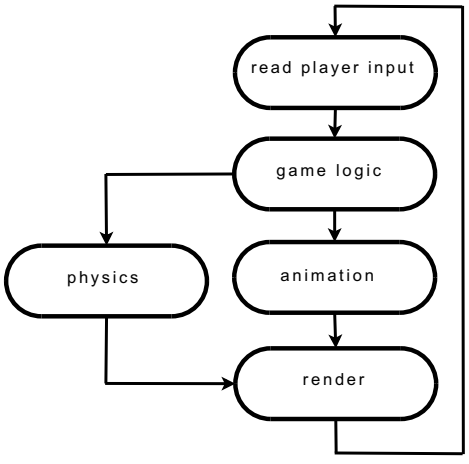

Figure 6 – Synchronous Function Parallel Model.

The author 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 [17] is the formalization of the idea found in [5]. This model does not present a main loop. Figure 7 illustrates the model.

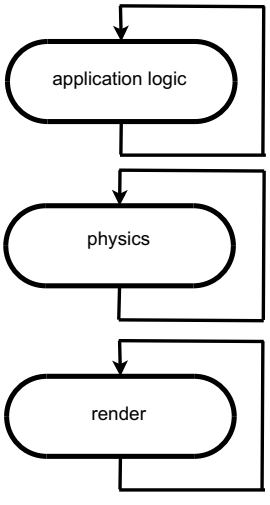

Figure 7 – Asynchronous Function Parallel Model.

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 [5].

Also Rhalibi et al. [23] 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 task ordering. In each thread, the tasks for rendering and update are divided taking into consideration their dependency.

The Data Parallel Model [17] uses a different paradigm in which data are grouped in parallel sections of the application where they are processed. So, instead using a main loop with concurrent parts that process all data, the Data Parallel Model proposes to use separate threads for data sets (like game objects). This way, the objects run their own tasks (like AI and animation) in parallel. Figure 8 depicts this approach.

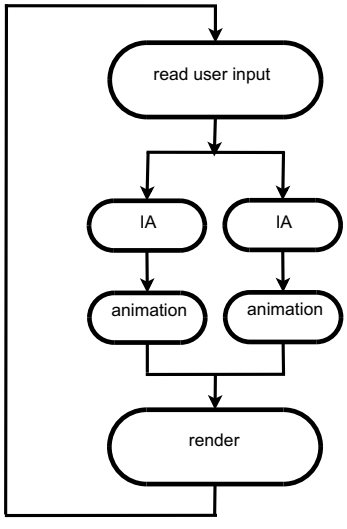

Figure 8 – Data Parallel Model.

According to the author, 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.

3 GPGPU

GPGPU or General-Purpose Computing on Graphics Processing Units (GPUs) is the use of GPUs for generic, non graphic, processing. Nowadays, GPUs are massively parallel architecture with more processing power than the CPUs. GPGPU has been the theme of research of diverse areas like: image analysis [12], linear algebra [2], chemistry [26], physics simulation [20], and crowd simulation [21], among others.

The first programmable graphic cards could only be programmed using low level languages (assembly). Afterwards, high level languages were created for GPU programming which were called shader languages. C for Graphics (Cg) was the first language and works with both APIs (DirectX and OpenGL). High Language Shader Language (HLSL) has been integrated with DirectX and the last one is OpenGL shader language (GLSL) which is part of OpenGL API. These languages were mainly designed for implementing graphics effects, but their usage on GPGPU problems was not trivial. Thus, GPGPU could be developed only by using either vertex or pixel shaders, with their limitations and idiosyncrasies [30].

The unified architecture changed the graphics cards architectures. There are not differences between vertex and pixel processors such as before. At the present, the stream processors are used for both vertex and pixel programs. Shader languages are still used, but mostly for graphics effects, and not for GPGPU programming anymore. For GPGPU, there are particular languages to work directly with the GPU. These architectures are used to process general data on the GPU providing: more flexibility on memory access; more integration with the traditional CPU languages and IDEs; usage without the integration with a graphic API; and mostly computational problems can be easier modeled for the GPU than using a shader or assembly language.

Nowadays, NVIDIA, AMD and Intel have been working in multi-core hardware and programming language solutions for GPGPU. NVIDIA developed Compute Unified Device Architecture (CUDA) [18] while AMD developed Compute Abstraction Layer (CAL) [1]. Both were based on the C language and work with an extension of C/C++ CPU language. Besides, Intel has worked with developing multi-core hardware based on x86 processors in parallel [25] and it is called Larrabee. Also there is OpenCL (Open Computing Language) [7] which is available for both NVIDIA and AMD graphics cards.

GPUs have been designed for problems that can be modeled as stream-based processes with intense mathematical calculations. The latency of memory has been a constraint in this field, since it can be the bottleneck of the simulation [14].

4 REAL-TIME LOOP MODELS WITH THE USE OF GPGPU

Real-time loop models are architectural solutions for managing and tidying tasks of the real-time applications. Users input, visualization and general processing, such as artificial intelligent, physics simulations, are instances of tasks computed in this loop.

With the development of the programmable GPUs, some of the traditional update steps of the real-time loop can be transferred from the CPU to the GPU, like artificial intelligence [24] and physics [19]. Hence, the traditional CPUs loops presented in section 2 must be adapted in order to take advantage of this new feature. This section presents the real-time loop models that use GPGPU available in the literature.

4.1 Single-Thread Real-time Loop Models with a GPGPU Stage

One of the simplest execution of the GPGPU step just after the execution of the rendering without any synchronization, can be seen on Figure 9.

This loop has been used in many available works [6, 20, 21]. This solution also could be adapted with a synchronization method to synchronize the loop to a fixed frequency as shown in Figure 10. This loop has the same disadvantage of possible waste of processing power as the Synchronized Coupled Model.

The time parameter also could be used to synchronize the loop without wasting processing power. This option can be seen on Figure 11.

The next subsections present some multithread real-time loop models with GPGPU. This kind of game loop involves parallel programming. Although there is an independent sections amount stages, the shared and non-shared parts must be detected, since they will be treated differently. GPGPU and update stages access the same parts, making necessary a synchronization object in order to guarantee mutual-exclusive access to shared data and preserve task execution ordering.

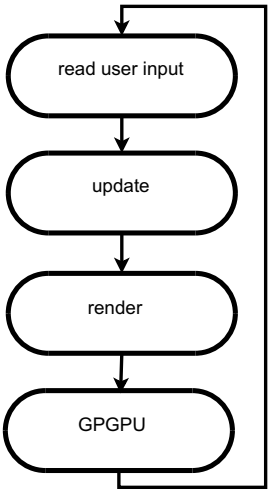

Figure 9 – Single Coupled Model with an GPGPU stage.

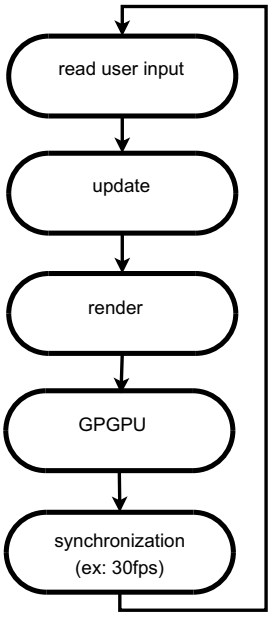

Figure 10 – Synchronized Coupled Model with a GPGPU stage.

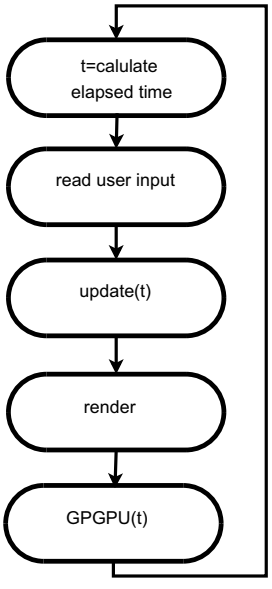

Figure 11 – Single Thread Uncoupled Model with a GPGPU stage.

4.2 Multithread Real-Time Loop Models with a GPGPU Stage Uncoupled from the Main Loop

This multithread architecture is composed by one thread for handling the user input, updating, and rendering stages, and another one for the GPGPU, as can be seen in Figure 12.

The literature [10, 11] presents two implementations of this loop model. The work describes an AABB (axis-aligned bounding box) collision detection implemented on the GPU and uses semaphores to synchronize the threads. The second work presents a hybrid physics engine which implements some of its methods

on the GPU. The synchronization was performed in the following way: the physics GPGPU loop was executed every 25 milliseconds and the main loop thread is updated as fast as possible, waiting for the result of the physics loop engine thread after every 25 milliseconds, as can be seen of Figure 13.

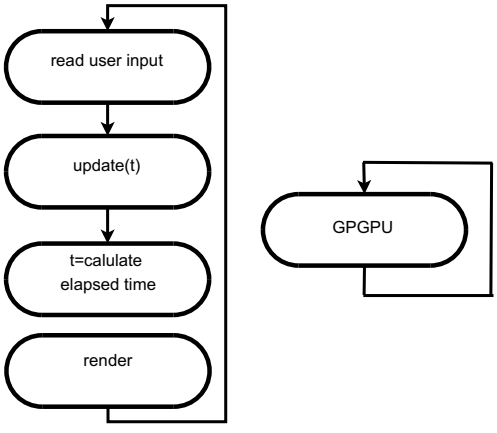

Figure 12 – The synchronism of the threads.

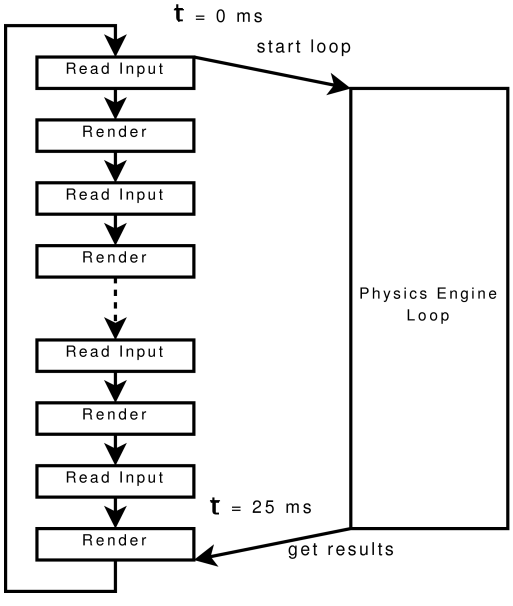

Figure 13 – The synchronism of the threads.

4.3 Multithread Real-Time Loop Model with a GPGPU Stage and the Render Stage Uncoupled from the Main Loop

The multithread uncoupled with GPGPU was presented in [29, 30]. It is a model based on the multithread uncoupled model (Fig. 3) with the inclusion of a new stage. There are three threads in this model, as can be seen on Figure 14. The first executes input and update stages. The second is responsible of GPGPU stage and the last one computes the render stage.

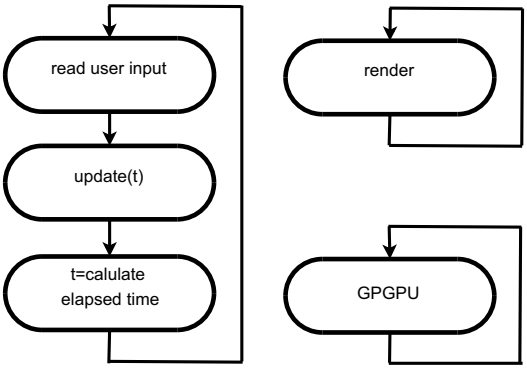

Figure 14 – Multithread Uncoupled with GPGPU.

As the synchronization must be done, GPGPU stage is executed after update and the synchronization happens between theses states, while the render stage runs in parallel to the previous ones. The update and GPGPU stages are responsible for defining the new simulation state. For example, the former calculates the collision response for some objects, whereas the latter defines new positions for the objects. The render stage presents the results of current application state. Multithread programming is a complex subject, as the tasks in the application run alternately or simultaneously, but not linearly.

5 REAL-TIME LOOP MODELS WITH AUTOMATIC DISTRIBUTION BETWEEN CPU AND GPU

With the development of models for running simulation tasks on CPU and also on GPU, come other models that can take the best characteristics of both environments. The motivations for employing automatic task distribution schemes are as follows:

1. Take advantages of the best characteristics of both the CPU and GPU;
2. Take out from the developer the decision of which processor should run some tasks, leaving this to the automatic distribution scheme to choose;
3. Redistribution of tasks between the processors when a processor (CPU or GPU) is overloaded with work.

In order to use a distribution between CPU and GPU the developer must provide both implementations and also mechanisms to change the processor without loss of information. Using this concept, it is possible to adapt the loop models from section 4 for the use with an automatic distribution of tasks, putting a distribution mode instead the GPGPU task to decide between the CPU or GPU to execute the task, as can be seen on Figure 15.

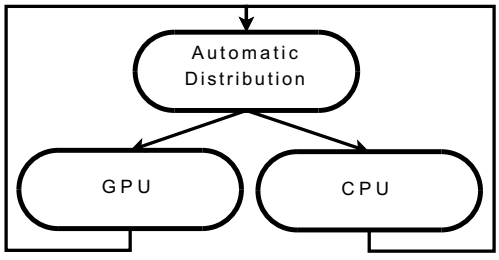

Figure 15 – The Distribution Module.

In the next subsections this work presents the forms of distribution: a manual distribution or an automatic distribution based on heuristics.

5.1 Manual Decision

The simplest decision on how to distribute tasks between the CPU and GPU is to let the user or developer to do it. This decision can be done during the application via a C function or even a script language, such as LUA [15]. Former works [29] implement a task scheduling distribution between CPU and GPU via a script file.

5.2 Heuristics for Automatic Distribution

The methods for automatic task distribution that were first presented in [10, 11] are: "starting" automatic distribution, "cycle" automatic distribution, "starting full test" automatic distribution, "adaptive" automatic distribution, "best time" automatic distribution and "resource" automatic distribution.

5.2.1 "Starting" Distribution

The "starting" strategy for automatic distribution between CPU and GPU is very simple: it computes a constant number of frames in the GPU mode and the same constant number of frames in the CPU mode. With the measured times, it selects the fastest processor to process all the frames of the application.

The reason why this method computes a constant number of frames in each processor is to avoid making a wrong decision that could happen if the scheme would only process only one frame at each one. All the subsequent schemes spend a constant number of frames in initial tests for each processor because of the same principle. Algorithm 5.1 illustrates this method.

This method, in normal conditions, always selects the fastest processor without the necessity for the user or developer to do it. This is important for the developer who does not know the hardware of the user of the application and wants to use the fastest processor available.

Even though the "starting" automatic distribution selects the fastest processor, it does not avoid the application from being slowed down by other processes of the system if the CPU or GPU is already overloaded with other tasks. Also, the simulated scene may change during the application, and with that, the fastest processor can no longer be the one selected from the "starting" automatic distribution.

5.2.2 "Cycle" Distribution

The "cycle" distribution has the following strategy: at every 100 frames, the engine calculates 5 frames in the GPU mode and more 5 frames in the CPU mode. From these times, the "cycle" automatic distribution chooses the fastest processor to simulate the next 90 frames, as can be seen on algorithm 5.2. In this case, only 5% of the frames are spent using the slower scenario and 95% with the best one.

This scheme is ideal for applications where the performance difference between both processors is low. If the difference between the processors is high, the 5% of the frames spent in the slowest mode can affect the overall performance of the application.

5.2.3 "Starting Full Test" Distribution

The "starting full test" is based on the same principle as the "starting" distribution, i.e, it also does an initial test in the beginning of the application but the difference is that this method does a full test.

This full test consists in the following strategy: it starts with the calculation of "minBodies" bodies simulation in 5 frames in the GPU and 5 frames in the GPU (represented by Execute5Frames in algorithm 5.3). Then it saves these times and increases the number of bodies multiplying by 2, and computes 5 frames in each processor, and so on until the "maxBodies" number of bodies is reached. "minBodies" and "maxBodies" are values that the developer must provide or use a default value (16 for "minBodies" and 8192 for "maxBodies"), like algorithm 5.3 illustrates. Based on those computed times, the simulation can determine which processor is faster for a determined scene.

This mode always selects the fastest processor for a given scene, but it is very intensive and spends considerable time in the beginning of the application. This method can prevent changes in the scene (number of bodies in the physics calculation) without the necessity of spending 5% of its frames in the slowest case.

Algorithm 5.1 "Starting" distribution

```
if frameCount == 10 then
  calculateElapsedTimeGPU()
  mode ⇐ CPU
else
  if frameCount == 20 then
    calculateElapsedTimeCPU()
    if GPUTime < CPUTime then
      mode ⇐ GPU
    end if
  end if
end if
return mode
```

Algorithm 5.2 "Cycle" distribution

```
if frameCount == 5 then
  calculateElapsedTimeGPU()
  mode ⇐ CPU
else
  if frameCount == 10 then
    calculateElapsedTimeCPU()
    if GPUTime < CPUTime then
      mode ⇐ GPU
    end if
  end if
else
  if frameCount == 100 then
    frameCount ⇐ 0
    mode ⇐ GPU
  end if
end if
return mode
```

Algorithm 5.3 "Starting full test" distribution

```
numBodies ⇐ minBodies
mode ⇐ GPU
while numBodies < maxBodies do
  Execute5Frames()
  if frameCount == 5 then
    calculateElapsedTimeGPU()
    mode ⇐ CPU
  else
    if frameCount == 10 then
      calculateElapsedTimeCPU()
      SaveTimes()
      frameCount ⇐ 0
      mode ⇐ GPU
      numBodies *= 2
    end if
  end if
end while
```

5.2.4 "Adaptive" Distribution

The "adaptive" distribution is based on the "cycle" distribution. The strategy is to perform an initial test in which 5 frames are computed on the GPU and 5 ones are processed on the CPU, and then to compute the next 90 frames on the fastest processor. After the execution of these 100 frames, the "adaptive" distribution starts to check every frame if the number of bodies has changed more than a percentage p since the initial test. When this happens, it does the initial test again to check which processor is faster for the current number of bodies and after the 100 frames it restarts to check the number of bodies again, as can be seen in algorithm 5.4.

Because of the characteristics of the processors, this mode tries to detect when the number of bodies increases (above a certain percentage p) if it is running on the CPU and when the number of bodies decreases (below the specified percentage p) if it is running on the GPU. The value of p must be provided by the developer, otherwise a default value of 20% is used.

Algorithm 5.4 "Adaptive" distribution

```
if frameCount == 5 then
  calculateElapsedTimeGPU()
  mode ⇐ CPU
else
  if frameCount == 10 then
    calculateElapsedTimeCPU()
    if GPUTime < CPUTime then
      mode ⇐ GPU
    end if
  end if
else
  if frameCount > 100 AND |numBodies − initialBodies| /initialBodies > p
  then
    initialBodies ⇐ numBodies
    frameCount ⇐ 0
    mode ⇐ GPU
  end if
end if
return mode
```

This mode in the best-case scenario behaves like the "starting" mode and in the worst case behaves like the "cycle". So, this mode is able to select the fastest processor for the physics simulation without the necessity of spending 5% of its frames in the slowest scenario (in the best case) and without spending considerable time with an extensive test on processors.

5.2.5 "Best Time" Distribution

The idea of this scheme consists in creating an approach that is able to redistribute tasks between CPU and GPU without the costs of making a lot of tests in the slowest processor, similar to the "cycle" distribution scheme.

This scheme presents this strategy: it starts calculating 10 frames on the GPU, and another 10 in the CPU. At the end of these 20 frames it determines the processor with the fastest time, and uses it for the next 10 frames. If the time of this processor after these 10 frames is less than the best time of the other processor, it calculates 10 frames on the other processor, and so forth. This scheme always saves the fastest time of each processor. Algorithm 5.5 implements this strategy.

The idea of this distribution is to use the fastest processor in most cases and to use the slowest one to take out work of the fastest processor when it is overloaded.

5.2.6 "Resource" Distribution

The idea of the "resource" strategy consists in the use of libraries to verify the processor usage (percentage).

The Windows API was used for this verification on the CPU. Initial tests have shown that the CPU usage varies with the number of bodies and remains constant when this same number does not vary, showing that this verification can be used for the purpose of distribution.

Algorithm 5.5 "Best time" distribution

```
if mode == GPU then
  if GPUBestTime > GPUTime then
    GPUBestTime ⇐ GPUTime
  end if
else
  if mode == CPU then
    if CPUBestTime > CPUTime then
      CPUBestTime ⇐ CPUTime
    end if
  end if
end if
if frameCount == 10 then
  calculateElapsedTimeGPU()
  mode ⇐ CPU
else
  if frameCount == 20 then
    calculateElapsedTimeCPU()
    if GPUBestTime < CPUBestTime then
      mode ⇐ GPU
    end if
  end if
else
  if frameCount > 30 then
    if mode == GPU then
      if GPUTime > CPUBestTime then
        frameCount ⇐ 21
        mode ⇐ CPU
      end if
    else
      if mode == CPU then
        if CPUTime > GPUBestTime then
          frameCount ⇐ 21
          mode ⇐ GPU
        end if
      end if
    end if
  end if
end if
return mode
```

The NVPerfKit [13] was used for checking this parameter in the GPU. The initial tests have shown that the GPU usage varies little with the number of bodies and do not remain constant when this number is constant, showing that is not possible to predict whether or not the GPU is overloaded with work by this method. Hence, this verification is not good for the purpose of distribution.

This scheme uses only the CPU charge, using the following strategy: 10 frames are calculated at the GPU and 10 frames at the CPU. Based on the obtained results, the fastest processor is selected, in the same manner that the "starting" automatic distribution. If the fastest processor is the CPU, it activates a variable "useCPU", and after 10 frames it starts to verify the percentage of use of the CPU, if this percentage raises to more than 80%, it sends the next 10 frames to the GPU. Algorithm 5.6 describes this approach.

This automatic task distribution scheme is aimed at simulations where the CPU is faster than the GPU, and the CPU uses the GPU as an auxiliary processor when it is overloaded with work. When the GPU is faster than the CPU, this scheme will not be able to redistribute tasks, and will behave like the "starting" automatic distribution.

Algorithm 5.6 "Resource" distribution

```
if frameCount == 10 then
  calculateElapsedTimeGPU()
  mode ⇐ CPU
else
  if frameCount == 20 then
    calculateElapsedTimeCPU()
    if GPUTime < CPUTime then
      mode ⇐ GPU
    else
      useCPU ⇐ true
    end if
  end if
else
  if useCPU AND frameCount > 30 then
    perc = getPercCPU()
    if perc > 80 then
      mode ⇐ GPU
      frameCount ⇐ 21
    end if
  end if
end if
return mode
```

6 A FRAMEWORK FOR REAL-TIME LOOP MODELS

Although processing power in consoles and computers has greatly increased, and multi-core architectures make it possible to use parallel processing, proper software is needed to extract high performance from the hardware. Even though the real-time loop models concept applies to both consoles and computers, there are some differences among these kinds of hardware.

Consoles (i.e. consoles in the same family such as the Xbox 360) have the same hardware, memory, processors and number of cores, making development in those platforms more predictable (i.e. the developers knows the hardware (s)he will be working with). On the other hand, for computers there is a myriad of configurations considering processors, memory, GPUs, and combination of these (and others) hardwares.

The proposed architecture works with multi-core CPUs and GPUs (if one is available). The architecture considers both as resources. A resource is a CPU core or a GPU, and the architecture encapsulates them. However, not all of loop tasks are suitable for processing both in the GPU and the CPU, as their architectures are different and require different programming paradigms.

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. 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 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 classes. 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 presented in this work. Also, the heuristics presented in section 5 can be adapted for this framework. An earlier version of this architecture was first presented in [9] 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. Therefore, the application has 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 can only run in the GPU, like the

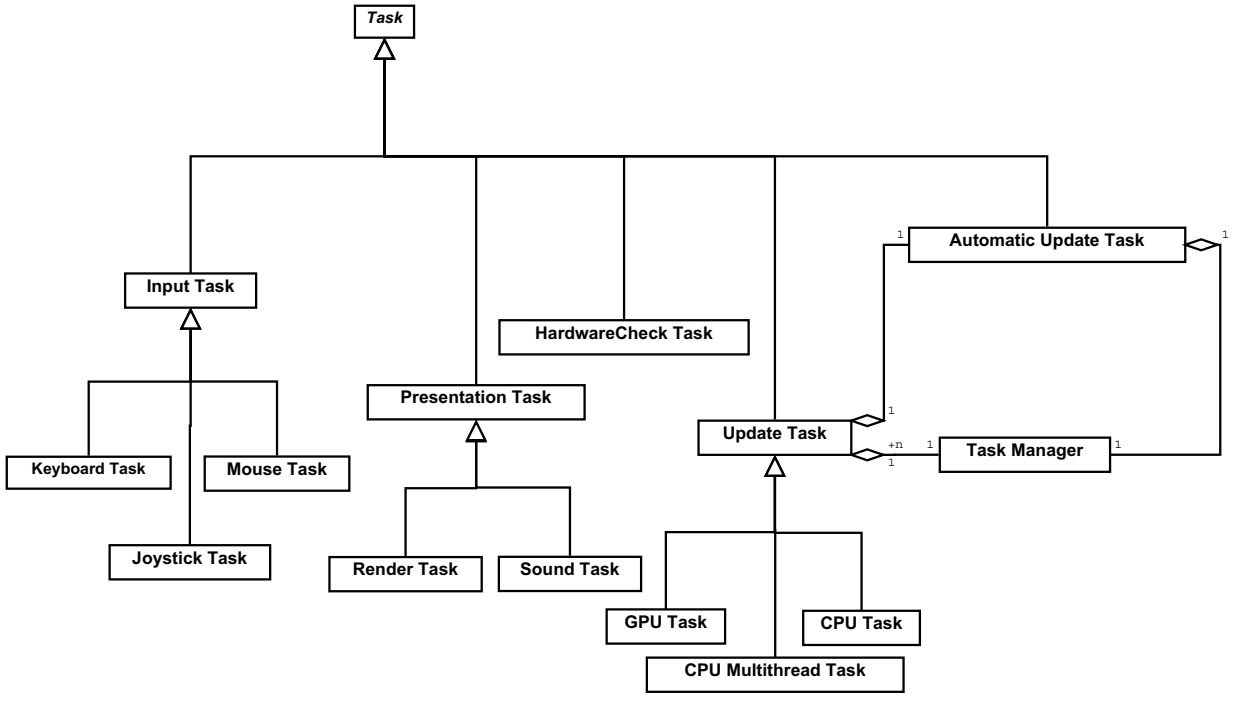

Figure 16 – Multithread uncoupled with an GPGPU stage.

presentation of the scene. The third group can also be modeled for running on both processors. These tasks are responsible for updating the state of some objects that belongs to the application, like AI and physics.

The task concept is modeled as an abstract class that different threads are able to load. Figure 16 illustrates the UML class diagram for the Task and its subclasses.

The Task class is a virtual base class and has five subclasses: Input Task, Update Task, Presentation Task, Hardware Check Task and Automatic Update Task. The first three are also abstract classes. The fourth is a special class to check the hardware. The latter is a special class whose work consists on performing the automatic dynamic distribution between the CPU and the GPU. This distribution consists of choosing the processor that is going to run a task according to some heuristic specified in a script file. Also a special class, the Task Manager class, is responsible for creating and keeping all the tasks of the loop (discussed in section 6.2).

The Input Task class and subclasses handle user input related issues. The Update Task class and subclasses are responsible for updating the loop state. The CPU Update class should be used for tasks that run on the CPU, and the GPU Update class corresponds to tasks that run on the GPU. The Presentation Task class and subclasses are responsible for presenting information to the user, which can be visual (Render Task) or audio (Sound Task).

6.1 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 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 in the end user computer. And with that information the automatic class can better distribute the task between CPU cores and GPU(s).

This checking 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 used.

6.2 The Task Manager

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. In order to con-

figure the execution of the tasks, each task has control variables described as follows:

- THREADID: an 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: an 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 that section 6.3 presents.

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 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. 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.

6.3 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 the following modes: one CPU thread only, multithread CPU, GPU only, and 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 presented in section 6.3.1. The scripts files are implemented in Lua [8] (section 6.3.2).

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 automatically 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 17 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.

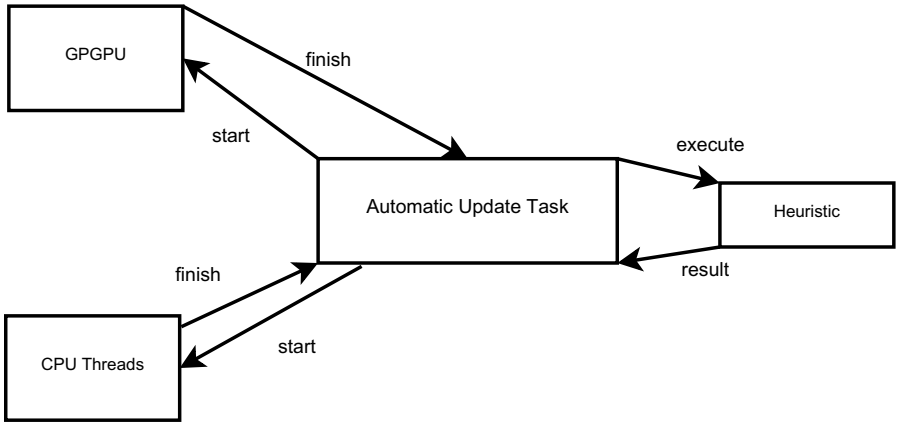

Figure 17 – The Automatic Update Task class and messages.

6.3.1 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 discard the first DISCARDFRAME frame results, because the main thread can be loading images or models and this can affect the tests;
- LOOPFRAMES: 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: 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 6.1.

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.

6.3.2 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;
  - 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 algorithm 6.2.
- 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 are executed.

Algorithm 6.1 Configuration Scritp

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

Algorithm 6.2 SetVariable Script

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

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.

7 TEST CASE AND RESULTS

The test case corresponds to a flocking boids implementation. The first well-know implementation of flocking boids was done by Reynolds [22] were it simulated a flock of birds with basic behavior rules. Each agent, also called boid, interacts with each other in a simple manner. The authors had implemented this example only to validate the framework of game loop proposed in this work. This example was implemented in single thread CPU, in a parallel CPU version and in a GPGPU version using CUDA.

The tests were performed in different computers with different hardware configurations to better illustrate the framework behavior. The hardware configurations are:

- Configuration 1: Intel Core 2 Quad 2.4 GHz CPU with 3 GB of RAM and equipped with an NVIDIA 8800 GTS GPU (a CUDA capable card) and the operating system is Windows XP 32 bits.
- Configuration 2: AMD Turion Dual-core 2.0 GHz with 3 GB RAM memory and equipped with NVIDIA 8200M GPU (a CUDA capable card), running on Windows Vista 32 bits.

For each configuration, the initial number of boids is 10 and it is increased until 2000 boids. The tests were made to run on CPU mode only, multithread CPU mode, GPU mode only (if available) and automatic mode. For heuristic of the automatic mode this work has used an adaptation of the "starting distribution" presented on section 5.2.1. The test results can be seen on Table 1 and Table 2.

These tests demonstrate that when there a few bodies, a single thread mode is faster than multithread modes (multithread CPU and GPGPU), but when the number of bodies increases the multithread modes are faster. The GPGPU mode is always faster, in the tests, than the multithread mode. But a multithread version is important in hardware configuration where a CUDA capable card is not available and the simulation requires a high number of bodies in the simulation. The tests also shows that the automatic mode always behaves like the best case.

8 CONCLUSION

Multi-core hardware architectures are a tendency. Recent CPUs now present quad-core processors and the GPUs present unified architectures to be used as GPGPU. Also video game consoles present multi-core processors (like the Xbox 360 and the Playstation 3). This tendency is not only in increasing the available processing power but also in rising the number processing units (cores). Hence, the use of parallel processing is a way to increase performance in real-time systems. For simulations that use GPGPU and multithread loop models, they take advantage on the available hardware and can present better performance. With that in mind, this work has presented a survey on real-time loop models for both single thread and multithread architectures with or without the use of GPGPU. It also presents a framework that makes possible to implement those real-time loop models, performing automatic task distribution across processors. Moreover, the framework can adapt itself in order to achieve the best

Table 1 – Performance rate of simulation in FPS with hardware configuration 1.

# Bodies	Configuration 1			
	Single Thread CPU	Multithread CPU	GPGPU	Automatic
10	5211	310	1412	1412
50	966	271	1334	1334
100	244	208	1236	1236
200	64	127	1098	1098
500	10	36	834	834
1000	2.6	9.8	586	586
2000	0.7	2.7	362	362

Table 2 – Performance rate of simulation in FPS with hardware configuration 2.

# Bodies	Configuration 2			
	Single Thread CPU	Multithread CPU	GPGPU	Automatic
10	138	92	111	138
50	131	89	107	131
100	142	77	105	142
200	46	50	102	102
500	7	16	95	95
1000	2.0	4.5	65	65
2000	0.4	1.2	32	32

performance on the available hardware, by checking the end user hardware configuration and selecting the best loop model for that configuration.

The automatic task distribution on real-time applications, like games and simulations, is very important because the application can make optimal use of the end user hardware. This causes a better performance in the end.

This work also remarks that the proposed framework for real-time loop models applies to others multi-cores hardwares like the Playstation 3 and the Xbox 360 (with some modifications).

REFERENCES

[1] AMD. 2007. Amd stream computing. Available at: http://ati.amd.com/ technology/streamcomputing/firestream-sdk-whitepaper.pdf. 20/02/2008.

[2] J. BOLZ, I. FARMER, E. GRISPUN & P. SCHRDER. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. ACM Transactions on Graphics, 22(3): 917–924.

[3] D. SANCHEZ & C. DALMAU. 2003. Core Techniques and Algorithms in Game Programming. New Riders Publishing.

[4] P. DICKINSON. 2001. Instant replay: Building a game engine with reproducible behavior. Available at: http://www.gamasutra.com/features/20010713/dickinson 01.htm/.

[5] H. GABB & A. LAKE. 2005. Threading 3d game engine basics. Available at: http://www.gamasutra.com/features/20051117/gabb 01.shtml/.

[6] SIMON GREEN. 2007. Gpgpu physics. Siggraph07 GPGPU Tutorial.

[7] KHRONOS GROUP. 2009. Opencl – the open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/opencl/.

[8] ROBERTO IERUSALIMSCHY, LUIZ HENRIQUE DE FIGUEIREDO & WALDEMAR CELES. 2006. Lua 5.1 Reference Manual. Lua.org.

[9] M. JOSELLI, M. ZAMITH, E. CLUA, P. PAGLIOSA, A. CONCI, A. MONTENEGRO & L. VALENTE. 2008. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, pages 115–120.

[10] M. JOSELLI, M. ZAMITH, L. VALENTE, E.W.G. CLUA, A. MONTENEGRO, A. CONCI, B. FEIJÓ, M. DORNELLAS, R. LEAL & C. POZZER. 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, pages 48–55.

[11] MARK JOSELLI, ESTEBAN CLUA, ANSELMO MONTENEGRO, AURA CONCI & PAULO PAGLIOSA. 2008. A new physics engine with automatic process distribution between cpu-gpu. Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, pages 149–156.

[12] A. KERR, D. CAMPBELL & M. RICHARDS. 2008. Gpu vsipl: High-performance vsipl implementation for gpus. In High Performance Embedded Computing.

[13] J. KIEL & S. DIETRICH. 2006. Gpu performance tuning with nvidia performance tools. Game Developers Conference.

[14] J. KRUEGER. 2008. A gpu framework for interactive simulation and rendering of fluid effects. IT – Information Technology, 4: (accepted).

[15] LUA. 2009. The programming language lua. Avalible at: http://www.lua.org, 2009. 13/01/2009.

[16] MICROSOFT. 2006. Xna 1.0 documentation. Available at: help:// MS.VSExpressCC.v80/MS.VSIPCC.v80/MS.XNAFX.1033/XNA/AppModel GameLoop.htm.

[17] V. MNKKNEN. 2006. Multithreaded game engine architectures. Available at: http://www.gamasutra.com/features/20060906/monkkonen 01.shtml.

[18] nVidia. 2007. Nvidia cuda compute unified device architecture documentation version 1.1. Avalible at: http://developer.nvidia.com/object/cuda.html, 2007. 20/12/2007.

[19] nVidia. 2009. Physx. Avalible at: http://www.nvidia.com/object/nvidia physx.html, 2009. 20/02/2009.

[20] LARS NYLAND, MARK HARRIS & JAN PRINS. 2007. Fast n-body simulation with cuda. GPU Gems 3 Chapter 31, pages 677–695.

[21] E. PASSOS, MARK JOSELLI, M. ZAMITH, J. ROCHA, A. MONTENEGRO, E. CLUA, A. CONCI & B. FEIJÓ. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, pages 81–86.

[22] CRAIG W. REYNOLDS. 1987. Flocks, herds and schools: A distributed behavioral model. In SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pages 25–34, New York, NY, USA, ACM.

[23] ABDENNOUR EL RHALIBI, STEVE COSTA & DAVID ENGLAND. 2006. Game engineering for a multiprocessor architecture. In DIGRA Conf.

[24] T. RUDOMN, E. MILLN & B. HERNNDEZ. 2005. Fragment shaders for agent animation using finite state machines. Simulation Modelling Practice and Theory, 13(8): 741–751.

[25] LARRY SEILER, DOUG CARMEAN, ERIC SPRANGLE, TOM FORSYTH, MICHAEL ABRASH, PRADEEP DUBEY, STEPHEN JUNKINS, ADAM LAKE, JEREMY SUGERMAN, ROBERT CAVIN, ROGER ESPASA, ED GROCHOWSKI, TONI JUAN & PAT HANRAHAN. 2008. Larrabee: A many-core x86 architecture for visual computing. ACM Transactions on Graphics, 27(3).

[26] IVAN S. UFIMTSEV & TODD J. MARTNEZ. 2008. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. Journal Chemistry Theory Computation, 4 (2): 222–231.

[27] LUIS VALENTE, AURA CONCI & BRUNO FEIJÓ. 2005. Real time game loop models for single-player computer games. In Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment, pages 89–99.

[28] J. WATTE. 2005. Canonical game loop. Available at: www.mindcontrol.org/∼hplus/graphics/game loop.html/.

[29] M. ZAMITH, E. CLUA, P. PAGLIOSA, A. CONCI, A. MONTENEGRO & L. VALENTE. 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, pages 37–43.

[30] MARCELO P.M. ZAMITH, ESTEBAN W.G. CLUA, AURA CONCI, ANSELMO MONTENEGRO, REGINA C.P. LEAL-TOLEDO, PAULO A. PAGLIOSA, LUIS VALENTE & BRUNO FEIJÓ. 2008. A game loop architecture for the gpu used as a math coprocessor in real-time applications. Comput. Entertain., 6(3): 1–19.