A Game Loop Architecture with Automatic Distribution of Tasks and Load Balancing between Processors

Blind for review

Abstract

Nowadays, multithread architectures for PCs (multi-core CPUs and GPUs), and game consoles (as Microsoft Xbox360 and Sony Playstation 3) is a trend. Hence, single thread games loops will not get the best performance on such architectures. For this reason, multithread game loops that take advantage of such architectures are gaining importance. There are a lot of multithread game loops that can be used in order to achieve better performance in a game, but they can not adapted for different architectures. This paper presents a new architecture for game loops that can detect and analyze the user hardware and adapts itself to a specific game loop that can achieve the best performance for that hardware.

Keywords:: Game loops, GPGPU, Task Distribution, Load Balancing, Real-time Systems

Author's Contact:

Blind for review

1 Introduction

Multi-thread architectures on PC are getting more and more common with the development of multi-core processors and the new GPUs architectures that can be used for generic processing. Also top of the line video games like the Microsoft Xbox 360 and the Sony Playstation 3 features multi-cores processors. With that, game architectures have to paralise and distribute its tasks between the processors, needing to utilize concepts from distributed and parallel systems in order to fully take advantage of the hardware. This work utilizes from these concepts like task distribution and load balancing adapting this concepts to the game loop architecture.

Games are interactive real-time systems and, like multimedia application, they have time constraints to execute all of its processes and present to the end user the results. If a game does not fulfil this requirement, it will lose its interactivity and consequently it will fail. A common parameter for measuring a game and simulations is in frame per second (FPS). The lower acceptable bound for a game is 16 FPS. Their 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 game some discard of the rendered frame may occur. In order to achieve the best FPS in a game, game loops are designed and developed.

Game loop is the structure that determinate the order that each task of the game is executed during the loop. The game loops is mainly divided in three categories: data acquisition, which gets the data from user's input; data processing, where the game logic are processed; and data presentation, where the results are presented to the end user though images and audio. There are several works that deal with game loops in order to achieve better results but normally they are very restricted to the designed hardware. This work present an framework to build game loops that can adapt to the user hardware in order to take the best performance from the hardware.

When a task is paralized in threads, the distribution of works between the threads sometimes it is not the ideal making threads needing to wait for the other(s). This is a common problem in distributed system, and to solve it a automatic distribution of the work between the threads is needed. This work implements this load balancing using a heuristic to solve this problem.

This work is an extension of the work [blind for review blind for reviewa], where a framework for game loops that can automatic distribute task between CPU and GPU and can also implement single or multi thread game loops. This work extended that work by presenting the following concepts:

- Automatic verification of the user hardware, and select which game loops will be used based on that knowledge;
- Load balancing between the threads of the work.

This work is organized as follows: Section two presents game loop background. Section three presents some related works. Section four presents and explains all the major functionalities of the proposed architecture. In section five six presents the conclusion.

2 Game Loops

The game loop is the underlying structure upon games are built. Games are regarded as real-time applications because this kind of application has time constraints to run the tasks that rely on them. This means that if those tasks do not run fast enough, the experience the game must provide will be compromised.

The tasks that a computer game 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: mice, joysticks, keyboards, and motion sensors. The data processing part refers to applying the user input into the game (user commands), applying game rules (game logic), simulating the game world (game Physics), simulating non-player characters (Artificial Intelligence), and related tasks. The presentation refers to providing feedback to the user about the current game state, through images and audio.

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

3 Related Works

Although the game loop represents the heart of computer games, there are not many academic works devoted to this subject. The works by blind for review et al [blind for review blind for reviewb], Dalmau [Dalmau 2003], Dickinson [Dickinson 2001], Watte [Watte 2005], Gabb and Lake [Gabb and Lake 2005], and Mnkknen [Mnkknen 2006] are among the few ones. All of them focus on single-player games. The simplest real time game loop models are the coupled ones. The Simple Coupled Model [blind for review blind for reviewb] perhaps is the straightforward approach to modeling game 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 comes to using it in different machine configurations. The uncoupled models separate the rendering and update stages, so they can run independently, in theory. A nave 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 Multi-thread Uncoupled Model [blind for review blind for reviewb] try to bring determinism to the game execution by feeding the update stage with a time parameter. Figures 2 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

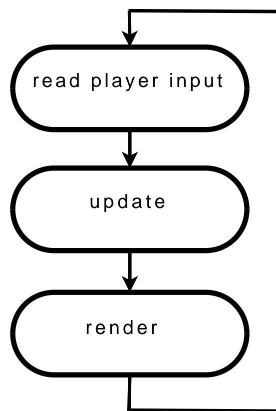

Figure 1: *Simple Coupled Model*

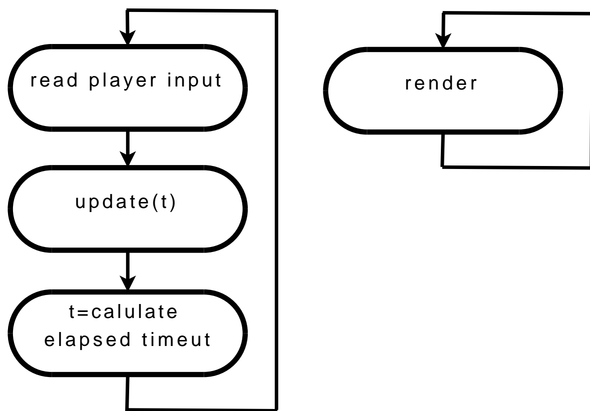

Figure 2: *Multi-thread Uncoupled Model*

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 games may require a scene replay feature [Dickinson 2001], which may not be trivial to implement if it is not possible to run some game sequence in a deterministic way. Other features as network module implementation and program debugging [Dickinson 2001] may be easier to implement if the game uses a deterministic model. Another issue is that running some simulations too frequently, like AI and game logic, may not yield better results.

Nowadays, computers and new video game consoles (such as the Xbox 360 and the Playstation 3) feature multi-core processors. For this reason, game loops that take advantage of these resources are likely to become important in the near future. Therefore, parallelizing game tasks with 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, not all tasks can be fully parallelized due to dependencies among them. As examples, 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 gaming computation. Mnkknen [Mnkknen 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 [Mnkknen 2006] proposes to allocate a thread to all tasks that are (theoretically) independent of each other. For example, performing Physics simulations while calculating animation. Figure 3 illustrates this 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

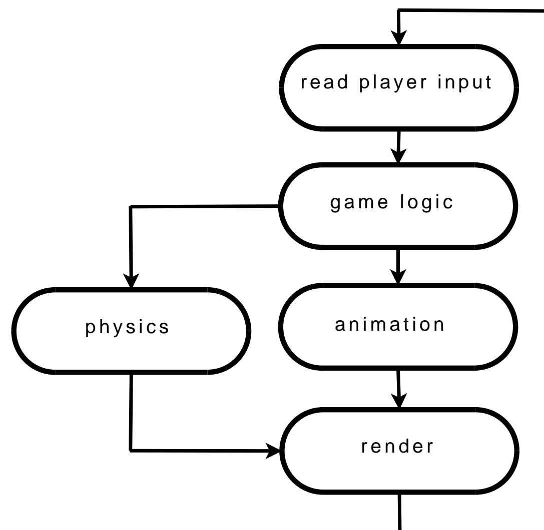

Figure 3: *Synchronous Function Parallel Model*

[Mnkknen 2006] is the formalization of the idea found in [Gabb and Lake 2005]. This model does not present a main game loop. Figure 4 illustrates the model.

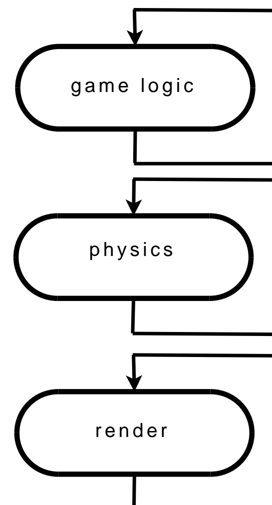

Figure 4: *Asynchronous Function Parallel Model*

Different threads run the game 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 game 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].

Also Rhalibi et al [Rhalibi et al. 2005] shows a different approach for game loops that is modeled taking the tasks and its dependency into consideration. Its divides the game 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 [Mnkknen 2006] uses a different paradigm where data are grouped in parallel sections of the application where they are processed. The Data Parallel Model proposes to use separate threads for sets of data (like game objects), instead of using a main loop with concurrent parts that process all data. In this configuration, the objects run their own tasks (like AI and animation) in parallel. Figure 5 depicts this approach.

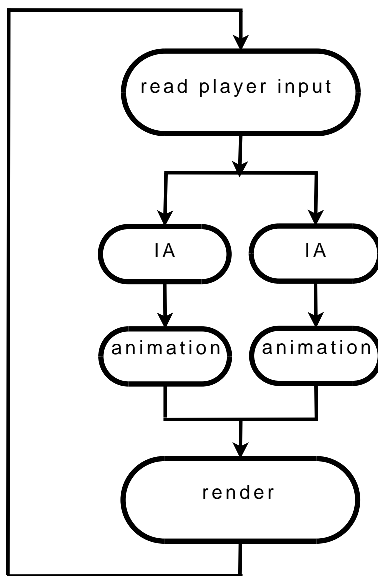

Figure 5: *Data Parallel Model*

According to [Mnkknen 2006] , this model scales well because it is able to allocate many processing cores as there 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. Mnkknen 2006 states that the biggest drawback of this model is the need to having components designed with data parallelism in mind. This type of parallelism has a similar approach as the GPGPU (general propose computation on graphics processor) work which also separates GPUs threads for sets of data.

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 Martnez 2008], physics simulation [Nyland et al. 2007], and crowd simulation [blind for review blind for reviewc].

There are some works that discuss using GPGPU with game loops [blind for review blind for reviewd], [blind for review blind for reviewe], [blind for review blind for reviewf], [blind for review blind for reviewg]. 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.

Blind for review [blind for review blind for reviewa] 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 Blind for review by proposing a game loop that is able to detect the available hardware and automatically distribute tasks among the various CPU cores and also to the GPU.

4 The Proposed Architecture

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 game loop concept applies to both consoles and computers, there are some differences among these kind 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 this (and other) hardware.

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 game 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 it is able to analyse 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 game loop is used in many games, 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 monitors the processors to calculate the usage rates of all of them.

4.1 The Task Class

A task is defined such as a work that the application (the game) should execute. There are three classic tasks that all game application execute: player input, game state update, and providing feedback to the player. These tasks should be broken in small subtasks, for example: Player input is performed by reading the keyboard, mouse or joystick state. Feedback to the player is provided by rendering the scene graphically, playing a sound effect, or vibrating the joystick. Hence, a task can be anything that the application works towards processing.

A constraint of this architecture is the fact that not all tasks can be processed by all sorts of processor architectures. An example is the CPU and the GPU, that have different hardware architectures. The GPU hardware is not able to access all of computer peripherals just like the CPU. For example: read a file from hard disk. It is a task that only the CPU is able to process. On the other hand, physics simulations can be processed by both, although GPUs typically execute it faster than CPUs.

Thus, a task is broken in three groups. The first group works with tasks that can be invoked only by the CPU in its cores, such as player input, file handling, and the managing of other tasks. The tasks of this group can be allocated in different cores, and, if there are more than CPU core, they can run in parallel. In the same way, the second group handles the tasks that can be only processed by GPUs, such as running shader processing. The last one is the group that can be modelled to be invoked by both processors. Figure 6 illustrates the UML class diagram for tasks.

The third group is the main focus of this work, because its tasks can be invoked by all processors. We are specifically interested in the issue of delegating a task to a CPU core or to the GPU.

The task structure is described as follows:

- TASK_ID: it is the id of the task. Whenever a task is created, it receives this unique number. The architecture uses the id to guarantee the correct order of execution and to identify which task is accessing the data shared among the tasks;
- TASKTYPE: the task type, that can be the following: input, update, presentation, and management (this last one identifies Task Manager and the Hardware Check classes);
- STACK: It is a stack area used to preserve the task state whenever it leaves the processor;
- DEPENDENCY: ID of next task that should run after this one. The Task Manager relies on this information to guarantee the correct execution ordering;
- PRIORITY: The Task Manager uses this value to decide which tasks in the task queue are going to be scheduled in each processor. The next Section provides more details about this field.

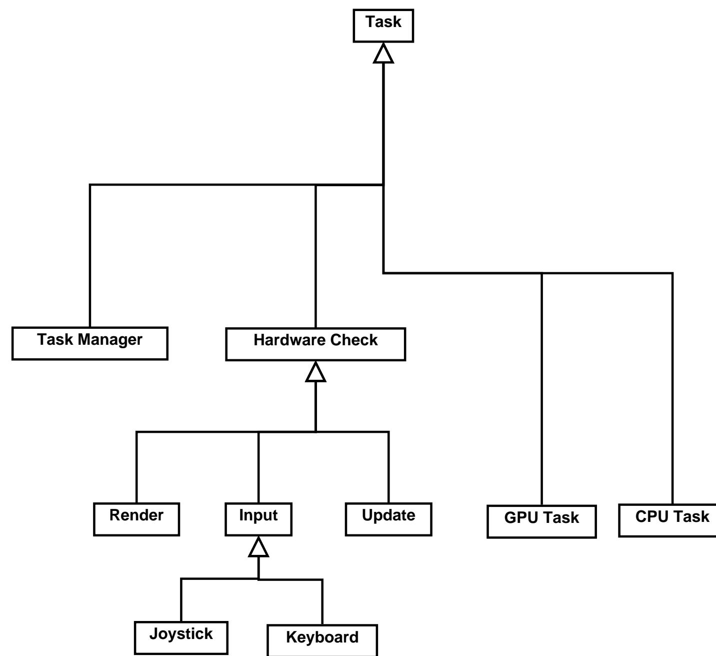

Figure 6: *UML class diagram*

- Load balancing between the threads of the work.

4.2 The Task Manager Class

The Task Manager is a special class derived from the generic Task class, as Figure 6 illustrates. It provides management, instancing, finalizing, and synchronization of all others tasks (except itself).

To guarantee the correct distribution and the abstraction of the tasks, the architecture defines a special task called Task Manager. The Task Manager is responsible for defining which task runs in which processor, and it should check whether a task can be broken in smaller tasks or not, and redistribute them. This class is derived from the generic Task class, and it provides management, instancing, finalizing, and synchronization of all others tasks (except itself). In order to guarantee central control of tasks and their correct execution, there is only one instance of the Task Manager class in the application, implemented as the Singleton design pattern.

The Task Manager class loads scripts that describe the initial policy of distribution that should be adopted. This means which messages it will send and the rules that should be applied. The input parameters of the script are: the task name, the elapsed time, and which processor is being used. The script has a task list and their constraints. The task constraints are: the execution ordering and which processor should be used.

For each kind of task, the Task Manager holds one version of the algorithm for the CPU (CPU Task) and another version (if applicable) for the GPU (GPU Task), both of them with the same distribution policy. The Task Manager creates instances of the GPU Task objects to run in the GPU thread. If there is a processor in the system with four cores, the Task Manager will create four instances of the CPU Task class, according to the script rules. Thread synchronization must be guaranteed in the algorithms themselves. The GPU Task objects work the same way. The Task Manager identifies how many GPUs are there in the system, and then it breaks the task among them.

To make it possible to use a parallel programming model, the Task Manager implements a multithread game loop. To avoid the problems of this model of programming, the Task Manager uses a semaphore such as the synchronize object. Accessing shared data, starvation and deadlocks are examples of parallel programming model problems. Figure 7 illustrates the game loop model this work proposes.

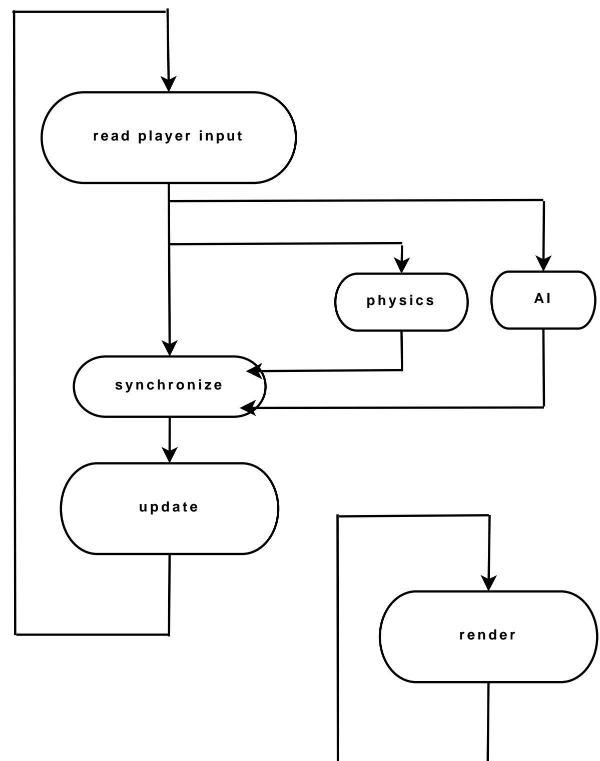

Figure 7: *Multi-thread Uncoupled Model (CPU/GPU)*

The Task Manager sees the CPU cores and the GPU as processing resources. To distribute best the application among these resources, the Task Manager exchanges messages with the Hardware Check objects (they are described in the next Section). These messages are exchanged during the execution of the application. The Hardware Check objects send information about the performance of the tasks to the Task Manager, which in turn uses this to decide about the best load balancing to apply.

Each task holds information that the Task Manager queries and updates. Recapping from Section 4.1, these information are: task id, task type, a stack, dependencies on other tasks, and task priority.

The tasks send messages to the Task Manager when these events occur:

- A task finishes execution;
- A task finishes processing shared data.

When a task finishes processing some shared data, the message it sends to the Task Manager is interpreted as a release of the shared data.

For each CPU core and each GPU, there is a processing thread and an associated task queue. When a task finishes executing, the Task Manager chooses the next task to run based on the priority values of the tasks in the queue.

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.

The Task Manager performs load balancing according to the usage rate of processors. Another class, named Hardware Check, is responsible for providing the information about usage rates. Section 4.3 describes this class.

4.3 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 keeps track of the number of CPU cores and GPUs available in the system. While the application is running, it watches each processor to calculate the usage rates. The class informs the Task Manager about the current state of the processors and their cores. The possible states are: ”waiting” and ”running”.

The class uses an ordered data structure to store information about the usage rates of each processor. In this case, using a simple vector is enough. Keeping the vector ordered guarantees that the first element will be that represents the highest usage rate.

To keep the Task Manager informed about the usage rates and processor states, it is necessary to establish a communication channel between them. Figure 8 illustrates this communication scheme.

4.4 Ways to Distribute the Tasks

The architecture perfoms the initial task distribution based on heuristics by reading a script file that contains the rules. This script file is written with the Lua language [R. Ierusalimschy 1996]. This script should not be complex to avoid delaying application startup.

The distribution of the tasks is done based on heuristics described and load before the execution of the application and it is written in LUA language . Despite the script of the heuristic is being loaded and processed during the start of application, it should not be complex to avoid spending time processing.

The architecture supports two kinds of heuristics. The first one is manual, which means specifying tasks for specify processor, ignoring performance. This heuristic is used to test task performance. The second one is the automatic heuristic.

The automatic heuristic should describe all tasks used by the application, their ordering and the types of processors the tasks use, as well as the rules to apply for changing processors. The elapsed time or frames per second are the only information the heuristic uses. It is a simple measure and can be applied for GPUs and CPUs. The input parameters are the processor ID and the time elapsed by the processor to process it. It returns the processor ID for the processor that should run the task after that execution.

The algorithm 1 is an example of an initial condition of the heuristic. It is configured to loop in the following state: execute 5 frames in the CPU and 5 frames in the GPU and decide to for the faster processor to execute the next 200 frames. It also specifies the number of CPU threads and the number of GPU threads to use. The script also specifies the number of frames to use when calculating the fastest processor and the measurement interval [blind for review blind for reviewa]. The number of CPU and GPU threads defines how many threads the application starts.

The architecture automatically distributes tasks between processor cores. If a core takes longer to process a task than other cores, the Task Manager removes some of the load of the heaviest task, the Task Manager removes some of the load of the heaviest task (i.e the task that is taking longer to run), and put in the lightest task (i.e the task that is taking less time to run).

Algorithm 1 INICIAL CONDITION

$INITFRAMES \Leftarrow 20$
$DISCARDFRAMES \Leftarrow 5$
$LOOPFRAMES \Leftarrow 50$
$EXECITEFRAMES \Leftarrow 5$
$CPUTHREAD \Leftarrow 4$
$GPUTHREAD \Leftarrow 2$

4.5 The Architecture Execution

This subsection is dedicated to illustrate the execution of the architecture, so the reader can better understand it. Figure 8 illustrates the process.

First if there are GPGPU tasks, the Hardware Check queries if there is a GPU card that can process the task. If this is the case, it informs the Task Manager that this task is able to run either in the CPU or the GPU. After that, Hardware Check queries how many CPU cores are available in the system, and pass this information to the Task Manager. The Task Manager, in turn, distribute the tasks through the available hardware.

5 Conclusion

The development and evolution of multi-cores processors, GPUs and video games indicates that multhitread architectures is a trend.

This work discussed the concept of game loops, a subject that is not very 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 the CPU and the GPU.

The proposed architecture is able to detect the available hardware, and then to break tasks into CPU cores and, if it is available, send them to the GPU.

References

BLIND FOR REVIEW. blind for review. blind for review. *blind for review*, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. In *blind for review*, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. In *blind for review*, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. *blind for review*, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. *blind for review blind for review*, blind for review, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. *blind for review*, blind for review.

BLIND FOR REVIEW. blind for review. blind for review. *blind for review*, blind for review.

BOLZ, J., FARMER, I., GRISPUN, E., AND SCHRDER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid.

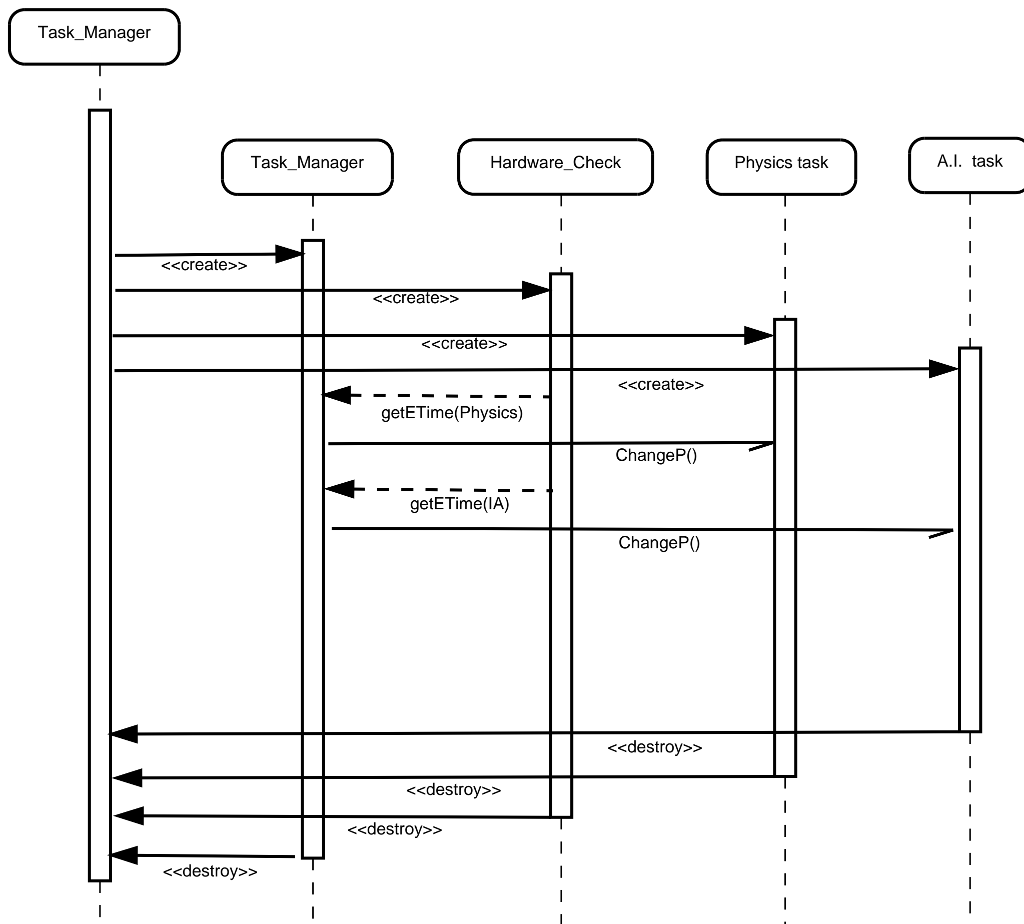

Figure 8: *Interation diagram*

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/ .

KERR, A., CAMPBELL, D., AND RICHARDS, M. 2008. Gpu vsipl: High-performance vsipl implementation for gpus. In *High Performance Embedded Computing*.

MNKKNEN, 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.

R. IERUSALIMSCHY, L. H. DE FIGUEIREDO, W. C. 1996. Lua - an extensible extension language. *Software: Practice & Experience 26*, 6.

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. 1. strategies for two-electron integral evaluation. *Journal Chemistry Theory Computation 4 (2)*, 222 – 231.

WATTE, J., 2005. Canonical game loop. Available at www.mindcontrol.org/~hplus/graphics/game_loop.html/ .