Parallel game architectures with tardiness policy and workload balance

Marcelo Zamith*
*Computer Science Department
Universidade Federal Rural do Rio de Janeiro
Nova Iguaçu - Brazil
Email: zamith.marcelo@gmail.com

Luis Valente†
†MediaLab/Institute of Computing
UFF, Niterói, Brazil
Email: lvalente,esteban@ic.uff.br

Bruno Feijó‡
‡VisionLab/Dept. of Informatics
PUC-Rio, Rio de Janeiro, Brazil
Email:bfeijo@inf.puc-rio.br

Mark Joselli§
§Escola Politécnica
Pontifícia Universidade Católica do Paraná - PUCPR
e-mail:mark.joselli@pucpr.br

Esteban Clua†

***Abstract*—Computer games are real-time applications that create interactive virtual environments, usually as discrete time-stepped simulations. These simulations may have predefined time step sizes or may use variable time step sizes. These approaches are common in games, but not flexible. In the first approach, when the game runs on a machine with abundant resources, the game does not use the extra capacity to improve simulation quality (task results or presentation). The second approach usually runs the simulation as fast as possible, using the time elapsed between consecutive time steps to scale all computations, so as the simulation runs in real-time. However, this approach wastes processor time and energy and in multi-core hardware scenarios (e.g., GPUs and clusters), the problem of wasting computing resources becomes more severe. In this paper, we propose a parallel and adaptive architecture that employs workload balance, precedence of game tasks and tardiness policy in multi-core hardware to handle the aforementioned issues. The architecture uses tardiness policy to monitor and change task behavior according to the current conditions of he host hardware. On more powerful computers, the architecture is able to improve task quality if there is spare time available. On less powerful computers, the architecture restricts task functionality so that tasks are able to complete on time. We provide two examples to demonstrate how the architecture works.**

***Index Terms*—heterogeneous and unconventional applications, tardiness, parallel programming, game architectures.**

I. Introduction

Computer games are real-time applications that create virtual environments, where players interact with the environment itself, with virtual characters, and with other players (in some cases). A game is regarded as a discrete simulation because the virtual environment is simulated by changing discrete game states across time. In this paper we consider *game state* as a static "data snapshot" that characterizes the game at any given moment. The simulation in a computer game usually advances in time using a time-stepped mechanism [1], [2].

Computer games provide the illusion that everything is happening at once, as smooth motions and immediate input feedback. This illusion is created by computing and presenting game states at a fast rate. The game state presentation takes form as images, animations, and audio, for example. In the time-stepped simulation model, each game state is computed in a single time step. The collection of time steps forms a game timeline. The game application processes each time step, sequentially. Usually, a game needs at least 16 time steps per second to maintain the game illusion, whereas more adequate values range from 30 and 60 [3], [4], [5]. We can affirm that games have real-time requirements because if a game application is unable to process all tasks before the current time step expires (i.e., the deadline), the user experience will not be good enough in fact, user experience could be severely impaired, thus breaking the "game illusion" The player may perceive a degraded user experience as jerkiness, unresponsive input, general slowness, and other undesired side-effects of low game performance.

Structurally, the game application uses a "game loop" to compute a single game state. A *game loop model* organizes the execution of game tasks to achieve consistent simulations[1], [2]. A game simulation is consistent if all tasks are processed in the correct order, considering their dependencies. For example, the game needs to collect player input before applying game rules. As another example, the game must run physics processes before repositioning non-player characters in the environment. In general, tasks in games can be categorized in three broad classes: input acquisition, simulation, and presentation. Input acquisition gathers player data from various input devices, such as mice, keyboards, sensors, and camera. Simulation corresponds to tasks that contribute to computing the current game state, such as game logic evaluation, applying game rules, physics simulation, and AI processing. Presentation tasks deliver simulation results (i.e., the current game state) to players through images (rendering), audio, and haptics, for example.

Game developers face a crucial challenge when implementing

game loop models in game architectures: it is necessary to satisfy real-time requirements in a scenario where hardware is highly heterogeneous. Desktop hardware has great diversity when considering CPU and GPU configurations. The hardware diversity increases when considering that computer games may use GPUs as high performance clusters to process massively mathematical problems, due to the high computing capabilities that these hardware provide for non-graphics tasks. Specialized video-game consoles also use diverse parallel hardware architectures [6], [7], [8]. With multi-core and parallel hardware becoming ubiquitous and even more affordable, parallel programming in game development is already a trend. The challenge of guaranteeing real-time requirements, consistency, and interactivity is not an easy task to achieve in practice when considering this scenario.

There are several ways to approach this challenge. For example, computer game architectures may have a well-defined simulation model or not. In well-defined simulation models (the first approach), developers determine a target time step size and design a game simulation based on this value. The simulation becomes deterministic and developers are able to reproduce a simulation given the same input data. Architectures that do not have well-defined simulation models (the second approach) commonly use variable time step sizes, which is a measure that renders the simulation as non-deterministic. In these cases, a common solution is to use the elapsed time between consecutive loops as the time step size.

Although these two approaches are common in game development, they are not flexible. In the first approach, when the game runs on a machine with abundant resources, the game does not use the extra capacity to improve the quality of task results or presentation. In the second approach, the game runs as fast as possible, wasting processor time and energy. When this second case takes place in scenarios with multi-core CPUs and GPUs, the problem of wasting computing resources becomes more severe.

Additionally, applications, including games, developed to multi-core CPUs brings another challenge that is the best utilization of parallelism on multi-core CPUs, i.e., all cores of a CPU must have the same quantity of workload in order to avoid idle state of some cores or processing overload on other one.

In this paper, we propose a novel game loop architecture that considers the positive aspects of the two approaches formerly described, while avoiding some of their shortcomings. We investigated a strategy employed in real-time simulations and applied it to game development to efficiently manage heterogeneous cluster architectures, as multi-core CPUs and GPUs for general purpose computing, adjusting the workload among the CPU cores. In other words, the proposed architecture builds a game simulation using time steps of fixed value and it is able to measure total task processing to request tasks to either reduce or raise their processing requirements according to the current hardware conditions. As far as we know, there are no research works in game development literature that apply the concepts as we present in this paper. The architecture we propose in this paper represents the first attempt at workload balance and applying tardiness control in parallel (CPU multi-cores and GPUs) game loop architectures.

The research that led to this paper had these main objectives:

1) Define the game task workload among available and the precedence and/or independence of game task;
2) try to optimize hardware usage according to game task workload;
3) develop a metric to describe task delays or advances given a target time step;
4) develop strategies to adapt task operation according the metric defined at (2); and
5) develop a game loop architecture that tries to ensure real-time requirements by applying (1), (2), (3) and (4).

The first point (1) aims at guaranteeing the uniform workload among CPU cores and taking into account the precedence and/or independence of game tasks, i.e., keeping all cores busy with the same quantity of workload, dividing all game tasks among CPU cores so that the game tasks can be executed in parallel and also in correct order. To define the game tasks order, we have adopted a heuristic to create the game tasks list and define how these tasks will be processed and allocate them on the processors available. We have used genetic algorithm (GA) as a heuristic[9].

Point (2) aims at maximizing hardware usage given the current run-time conditions. When total task processing time is below a defined target time step, there is room for improvement regarding task result quality. For example, in this situation an AI algorithm might try an alternate approach that uses more computing power to yield more sophisticated NPC behaviors. On the other hand, if total task processing power exceeds the target time step, tasks need to use less computing power so that game processing fits the target time step. In order to realize point (3), we have used a technique from real-time operating systems known as tardiness control [10]. Tardiness is a technique used as a metric to calculate task delays or earliness given a target time step, helping applications to satisfy real-time requirements. To realize point (4) we use the concept of interruptible and non-interruptible tasks, which we discuss in Section III. Finally, we apply points (1), (2), (3) and (4) as an adaptive and parallel game loop architecture that uses GA to guarantee the workload on CPU, and game loop architecture also uses tardiness policy to request tasks to adapt their operation, as a measure to try to meet real-time requirements and maintain game interactivity.

This paper extended the previous one presented in XIV Brazilian Symposium on Computer Games and Digital Entertainment. In this work, we include and present a preliminary study about game tasks list scheduling due to this research is being conducted.

This paper is organized as follows. Section II presents works related to list scheduling, tardiness and game loops. Section III presents the proposed architecture. Section IV demonstrates the proposed architecture through two case studies. Finally, Section V presents the conclusions and future works.

II. Related work

Real-time operating systems have more strict real-time requirements than games have. This means that if system does not execute all tasks before the deadline, the system fails, which may result in drastic consequences. For example, if these systems fail, people may die. Examples of these systems include aircraft control systems.

Generally speaking, research works in real-time operating systems use tardiness to propose diverse task scheduling strategies to avoid task delays, which can be fatal is those systems. Examples of such strategies are [11], [12], [13], [14], [15], [16], and [17].

Although researchers have been discussing tardiness control in real-time operating systems for a long time, we were not able to find research works that apply tardiness control in game simulations.

Regarding game loop architectures and game loop models, Valente et al. [18] provide an overview of basic game loop models. For example, the simplest game loop model consists of running player input, simulation, and rendering stages sequentially, as a pipeline (single-threaded). These kinds of models couple the simulation and rendering stages, which run as fast as possible. Consequently, the simulation does not satisfy real-time requirements as the simulation behaviour depends on the host hardware capacity. The simulation runs faster in powerful machines and slower in machines with limited computing power. A classic solution to solve this problem corresponds to uncoupling the simulation and rendering stages. Usually, this solution takes form as using elapsed time between consecutive loop executions as parameter in tasks calculations (e.g., physics and animations). By using this solution, the simulation runs in a similar way in different machines while maintaining interactivity. 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. However, this approach wastes processing power as the game runs as fast as possible. There are some models in the literature that apply this approach, in single-threaded ([18], [19]) or multi-threaded ([18], [20], [21]) variants.

As hardware evolved over time (as multi-core CPUs, programmable GPUs and distributed environments), game developers needed to design new approaches to take advantage of these hardware resources through parallel and distributed programming. There are a number of works in the literature that explores this issue (such as [22], [23], [24]), which is related to the solution we present in this paper.

Game architectures that use parallel and distributed computing need to address problems such as data sharing, data synchronization, and deadlocks. Another important issue is that some tasks cannot be fully parallelized due to task dependencies [20]. For example, the game is unable to render a character in the correct state before computing the game logic and updating the overall game state. Hence, serial tasks represent a bottleneck to parallelizing game simulations.

Considering task dependencies, Rhalibi et al. [25] present an early attempt to handle this issue as a game loop model that divides tasks into three concurrent threads, creating a cyclic-dependency graph to organize the ordering in game related processing. Each thread divides the rendering and update tasks according to their dependencies.

Mönkkönen [21] presents multi-thread game loop models that are grouped into two categories: function parallel models and data parallel models. The first category corresponds to models that present concurrent tasks, while the second category concerns models that try to process data entirely in parallel, if possible. As an example (first category), Mönkkönen proposed the Asynchronous Function Parallel Model, which does not wait for task completion to perform its job. The Asynchronous Function Parallel Model runs the render stage using the last complete game state, even if the update stage is still computing the new one. As an example related to the second category, there is the Synchronous Function Parallel Model [21], which processes the game physics in a separated thread while the main thread processes the characters animations.

There are some works that use the GPU in game loop architectures for non-rendering tasks (as GPGPU). Zamith et al. [23] proposed the first game loop architecture that uses GPUs as math co-processors in games. The work by Zamith et al. [23] extended the Single-thread Uncoupled Model [18] by creating an architecture that uses a secondary thread responsible for managing the GPU as a math co-processor.

Another work that applies the GPU for parallel programming is [22], where the authors present an architecture that distributes tasks between CPU and GPU. This architecture uses GPGPU to process game physics (in a dedicated thread) and implements static load balancing (the task distribution is determined by scripts before application start up).

Joselli et al. [24] also present an architecture based on a model that integrates GPGPU in the game loop. The Multi-thread Uncoupled Model proposes an automatic load balancing scheme that uses heuristics to define task allocation on processors (considering hardware with multi-core CPUs and programmable GPUs). This load balancing scheme is able to work dynamically, moving tasks between processors during the application lifetime to guarantee task load balance.

AlBahnassi et al. [26] propose a design pattern for parallel programming of games that support the creation of task graphs. This design pattern takes into consideration task heterogeneity, task dependencies and dynamic sets of active tasks. The

approach by AlBahnassi et al. [26] aims at maximizing multi-core CPU usage.

Best et al. [27] propose a parallel programming environment for games (named as Cascade) that supports task dependencies through dependency graphs. Cascade also implements a parallel version of the producer/consumer pattern, which is very common in games. Best et al. [27] also propose CDML (Cascade Data Management Language) as a solution to handle issues related to side effects in parallel procedural programming.

III. The Proposed Game Architecture

Digital games need to process simultaneously a set of heterogeneous tasks in a heterogeneous hardware, which means that it is not possible to predict the time required to process all the tasks. Consequently, it is not possible to predict the behavior of a game simulation, in terms of interactive performance.

To help in dealing with these issues, we propose an architecture that creates a list of game tasks and defines the cores workload in according to this list. Following, the proposed architecture also defines a policy to help the game simulation in meeting performance metrics.

A. The game tasks list scheduling

List scheduling problem is frequently discussed in combination optimization field. It is classified as NP-Hard problem and consists in finding the best tasks arrangement in order to minimized makespan. Makespan represents the amount of time required by an application to be completely processed [28].

Furthermore, our work focus on parallel multi-core architecture. It means, all cores same equal, all of them have the same power processing and can communicate data in low latency through cache memory hierarchy. The list scheduling is based on deterministic tasks. A deterministic list scheduling problem is a class of problem where we know everything about tasks, precedence, dependence, execution time and communication [28]. These characteristics represent game tasks due to the fact that game tasks are well behavior. We know about the moment that game task starts and how long it takes.

In case of games, list scheduling is a list of game task set which should be computed, otherwise, the game could fail. Furthermore, some tasks rely on player events. In this case, those tasks can occur or not in according to the player event. Therefore, game list scheduling considers all those aspects and constraint of game tasks.

The formal definition of list scheduling problem is: given by tuple (P, G), where a set of m homogeneous processors (cores of a CPU) $P = \{p_i : i = 1, 2, 3, \cdots m\}$; and a directed acyclic graph (DAG) $G = \{V, E\}$, where each vertex $V = \{T_1, T_2, \cdots, T_n\}$ represents a game tasks. The set of edged E means dependence among the tasks. Each $e_{i,j}$ connects the tasks T_i and T_j, the arrow of edged $e_{i,j}$ defines the processing order, which T_i is predecessor of T_j and T_j is successor of T_i. The successor task must not processed before all predecessors tasks have already been processed and their results are available. If a successor task is not ready to be processed, the processor is idle state until the task becomes ready.

Therefore, the list scheduling aim is find out the best arrangement possible throught a list, where all tasks are executed in parallel on all available processors, minimizing processors idle state and reducing the makespan.

We adopt DAG as data structure to represent game tasks relation, because it represents the dependencies among game tasks as well as their ordering. This data structure also provides us a way to represent communication cost among tasks, that given by edged weight [29], where edged weight 0 indicates no communication among tasks, while tasks that share data receive cost 1.

1) Genetic Algorithm: Genetic Algorithm (GA) is a heuristic inspired by theory of biological evolution proposed by Charles Darwin [30] . This heuristic is well applied in NP-hard problems and present good results in acceptable time. In most of case, the result is the best one. GA adopts the principles:

1) a problem solution is represented by individual;
2) the individual characteristics are his chromosomes that represent how adapted an individual is;
3) hereditary characteristics are transfer from individuals to their successors; and
4) natural selection determines that the most adapted individuals are chosen.

We may describe as following Algorithm:

Algorithm 1 GA to create a list of tasks

```
1: CreatingInitialPopulation(population);
2: EvaluatingPopulation(population);
3: while Stopping Criterion do
4:     if ( thenStopping Criterion = false)
5:         return solution;
6:     end if
7:     Crossover(population);
8:     Mutation(population);
9:     EvaluatingPopulation(population);
10:    DiscardingPopulation(population);
11: end while
```

The GA aim is disposing less adapted individuals and propagate the good individual characteristics, so that GA may find the best individual who is the best problem solution.

We code an individual as a set of list, where each list corresponds game tasks executed on a processor and order of tasks in the list indicates the order of execution. Each list

represents list scheduling of a processor. Number of lists is the same of number of processors.

Furthermore, two conditions should be satisfied:

- The precedence relations among the tasks are guaranteed.
- Every task is present and appears only once in the schedule [31], [32].

Figure 1 shows an example of GA individual, composed by 3 processors where the tasks are defined as DAG. Values over each task represents the elapsed time unit by the task, when it is executed on processor.

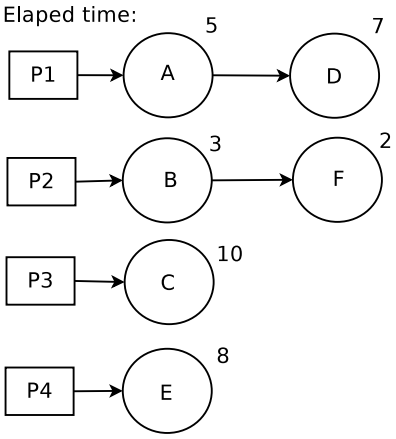

Fig. 1: *An example of A GA individual.*

Those tasks are allocated among processors. However, an individual can be a good or bad solution, relying on how the tasks are allocated. Fitness function uses the DAG to evaluating an individual. Fitness function provides values close to zero indicates that evaluated individual is good. Otherwise, values apart from zero indicates a bad individual or a bad solution. Indeed, fitness function tries to minimize makespan. The fitness function is given by equation:

$$makespan : min_{S_i \in Sched}\{max_{J \in Tasks} F_j\} \qquad (1)$$

F_j denotes the time when task j finalizes. $Sched$ represents the set of all possible schedules and tasks is the set of all game tasks are scheduled. Makespan of individual represented in Figure 1 is 12 time unit.

GA uses fitness function to select a set of individuals and discard other set. In our case, GA picks up the best individuals. These selected individuals indicate that they have good chromosomes. In doing so, good chromosomes will produce new best individuals. Otherwise, GA discards bad individuals.

We can decided to applied crossover operator in just one point, as a first approach. However, others crossover strategy can be applied. In our proposed work, GA selects two individuals and applies crossover operator in one point, as illustrated by Figure 2. After crossover operator, mutation selects individuals and changes two chromosomes.

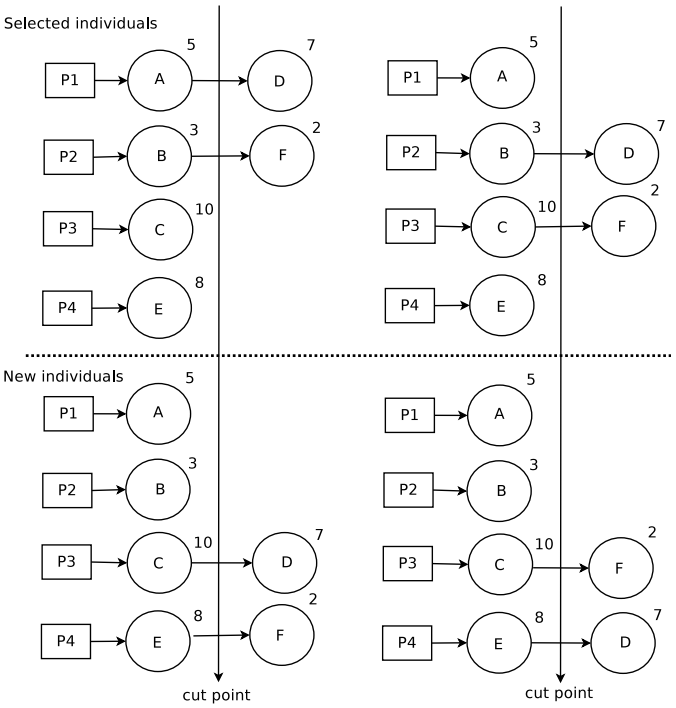

Fig. 2: *An example of two individuals generated by two other individuals by crossover operator.*

GA generates list scheduling with game tasks of game stage. For each new set of game tasks which executes concurrently, GA creates a new list scheduling. Given a list scheduling, the tardiness policy is applied, as we discuss in next sub-section.

B. Model Architecture

In a nutshell, the architecture measures elapsed time among consecutive loops and uses this value to verify if the simulation meets one of these situations: 1) the tasks are delayed; 2) there is processing time available; 3) the tasks are on time. In the first situation, total processing time is longer than the target time step value. In the second situation, total processing time is less than the target time step value. The third case is the optimal situation, would be having total processing time the same as the target time step value. Figure 3 illustrates these situations. The architecture uses the time measurements to adapt task execution to try to achieve the optimal solution, where total execution time equals the target time step.

The proposed architecture comprises three broad stages: input gathering, simulation, and presentation. The simulation stage runs tasks in parallel, according to the number of available processing cores. Figure 4 illustrates the proposed architecture, where the simulation stage is broken down into four sub-steps. The presentation stage corresponds to rendering.

The simulation stage lays out the four sub-steps (Figure 4) as follows. The first step is responsible for calculating the tardiness metric. The second step identifies how many tasks need to be processed and creates (n) threads to process them.

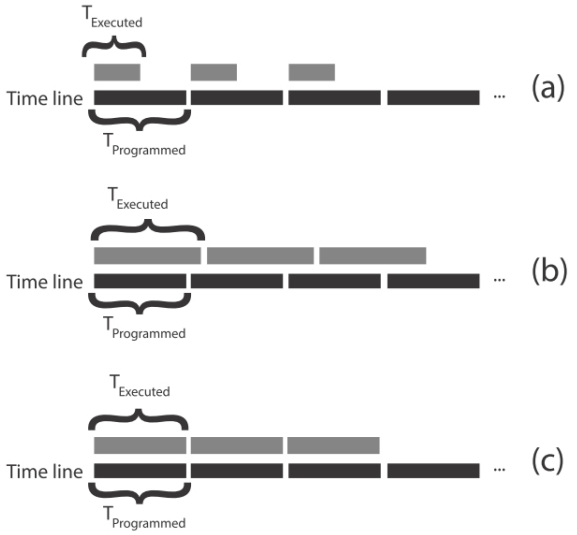

Fig. 3: *Task execution diagram. (a) Task finished before deadline: the available computing resources exceed task processing requirements (b) Delayed task (c) Optimal case.*

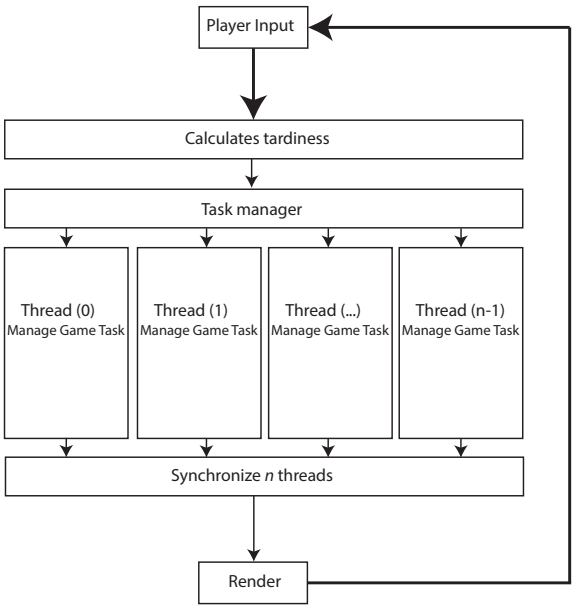

Fig. 4: *The proposed game loop architecture.*

The number of threads that the architecture creates depends on the number of available processing cores. At the third step, each thread runs its corresponding task, and at the fourth step the architecture synchronizes all threads to ensure that the results are consistent for presentation.

The developer is responsible for defining the granularity of game tasks. Granularity is the amount of work executed by a specific thread. Task granularity should be carefully chosen, since fine grained granularities may cause high system overhead due to thread management cost. On the other hand, coarse grained granularities may increase the thread workload, which may result in reduced parallelism levels and loss of thread processing efficiency. For example, a particle system requires processing several individual elements (i.e., the particles) and the developer may choose to process a particle system as a whole in a thread (using an iterative approach), or divide the particle system into sub-tasks to be processed in several threads.

C. The tardiness policy

The architecture applies tardiness control considering a target time step defined by the developer when designing the game. The target time step generally ranges from $1/30$ to $1/60$ seconds, which is a common value to keep interactivity in simulations and games. If total task processing time exceeds the target time step, the architecture may request a task to reduce its processing requirements or may try to interrupt the task's processing on the next time step. If the total task processing time is below the target time step, the architecture informs all the tasks that they may raise their processing requirements if needed. This brings opportunities for tasks to improve their results.

The architecture uses two kinds of tasks: interruptible and non-interruptible. Interruptible tasks may be broken down in several parts that are processed in consecutive time steps. When the target time step deadline comes, the architecture requests this kind of task to pause processing, which will be resumed in the next time step. The architecture repeats this procedure until the task completes. Non-interruptible tasks cannot be processed in consecutive time steps and cannot be interrupted. When the execution of a non-interruptible exceeds the target time step size, the architecture requests the task to reduce its processing requirements for the next time step. For example, a task may fulfil this goal by reducing the domain size of the problem that it needs to compute.

D. Tardiness metric

Our architecture uses tardiness as a metric for gauging system performance. In the first step (*Calculate tardiness*) the architecture considers the last elapsed time (previous loop execution) to calculate the tardiness value. The architecture uses this value to define a new time upper bound to keep total task processing time below the target time step (Eq. 2).

$$Tardiness = \frac{Time_{Executed}}{Time_{Programmed}} \tag{2}$$

Eq. 2 has two parameters: $Time_{Executed}$ (the measure task processing time) and $Time_{Programmed}$ (the predefined target time step). The developer is responsible for defining the target time step, which generally ranges from $1/30$ to $1/60$. This value usually matches the refresh rate of computer monitors. Eq.2 returns values for tardiness that range from 0 to ∞.

When the result is $0 \leq Tardiness < 1$, it means that total processing time is below the target time step. This situation brings opportunities to improve task quality due to extra processing time available. When Eq.2 results in values greater than 1 , it means that total processing time exceeded the target time step. In this situation, it is necessary to reduce task processing requirements to suit the target time step. The optimal case corresponds to Eq.2 resulting in 1, which means that total task processing time is exactly the same as the target time step. However, in practice the optimal case is hard to achieve. The proposed architecture tries it best to achieve the optimal case by using the tardiness metric ($Tardiness \approx 1$). Figure 3 illustrates these scenarios.

Eq. 2 is based on the original tardiness equation by Taranti et al. [10]. The original equation may provide negative values, which would prevent defining correct upper bound on time in our game architecture. Therefore, we have adapted the original equation to return values from 0 to ∞.

When considering several tasks, it is necessary to calculate the sum of all task processing times to compute tardiness. In this regard, $Time_{Executed}$ is given by:

$$Time_{Executed} = \sum_{j=1}^{n} GT_j \tag{3}$$

where GT_j denotes the processing time of the j^{th} game task.

Eq. 3 applies to calculating processing time of all tasks in a single processing thread. When there are several processing threads (the case of our architecture), the final $Time_{Executed}$ value will be determined by the thread that takes longer to process, because our architecture needs to wait for all threads to finish before proceeding to rendering. Eq. 4 represents this calculation:

$$Time_{Executed} = max\left(S_1, S_2, \cdots, S_n\right) \tag{4}$$

where S_i represents the total task processing time of the i^th thread.

We lay down the final equation to calculate total processing time in multi-thread contexts by combining Eq. 3 and Eq. 4 into:

$$Time_{Executed} = max\left(S_i\left(\sum_{j=1}^{n} GT_j\right)\right) \tag{5}$$

As a simple example of calculating tardiness (Eq. 2), let it be the following scenario. The target time step $Time_{Programmed} = 1$ms. The system has three different game tasks: 1) A particle system, such as gunshot smoke simulation (processing time $GT_0 = 0.5$ms); 2) AI behaviour of an enemy NPC "A" (processing time $GT_1 = 0.25$ms); 3) AI behaviour of an enemy NPC "B" (processing time $GT_2 = 0.15$ms); and 4) Calculation of player avatar movement (processing time $GT_3 = 0.3$ms). Consider that in this example there are three processing threads: thread A (particle system), thread B (AI behaviour of NPCs), and thread C (player movement).

In this case, $Time_{Executed} = max(0.5, 0.25 + 0.15, 0.4)$, i.e., $Time_{Executed} = 0.5$ms (Eq. 5). Applying Eq. 2 yields $Tardiness = 0.5$, which means that the total task processing time ($Time_{Executed}$) spent 50% of the target time step ($Time_{Programmed}$). As the tardiness value is below 100%, the game could use the extra time to the improve quality of some tasks, if desired.

As another example, suppose that the particle system in the previous example spent 1.25ms instead of just 0.5ms. In this case, $Time_{Executed} = max(1.25, 0.25 + 0.15, 0.4)$, i.e., $Time_{Executed} = 1.25$ms (Eq. 5). The tardiness in this example would be 1.25, which means that total processing time exceeds the target time step in 25%.

In both examples, the Task Manager adjusts game tasks as we explain in the next subsection.

E. Task Manager

The task manager is the core of our architecture. The task manager has two main responsibilities:

1) create and destroy threads dynamically, allocating tasks to threads (according to the task granularity defined by the developer); and
2) request tasks to adapt their processing time requirements according to the tardiness metric.

In our architecture, tasks need to address some requirements so that the architecture works properly. These requirements are:

1) a task is interruptible or non-interruptible — we define this as the "task behaviour";
2) a task is able to inform its task behaviour to the task manager; and
3) the task implementation uses time measures to adjust how it operates. According to task performance (evaluated with $Tardiness$), the task manager requests a task to reduce its processing demands, in case of $Tardiness > 1 + \delta$. The task manager also signals a task that it is allowed to raised processing demands if desired, in case of $Tardiness < 1 - \delta$. Our architecture uses the δ value to compensate for time interferences generated by background operating system tasks, which could lead to inconsistencies in our model. Currently, we use $\delta = 0.2$;
4) if the task is non-interruptible, the task implements methods to adapt its operation based on the tardiness metric; and
5) if the task is interruptible, the task implements methods to adapt its operation to be processed across consecutive time steps.

F. Implementation details

We defined an object-oriented model (C++ language) for our architecture where there are two main classes: a class to represent the task manager and a basic class to represent general game tasks. The basic task class defines two methods (*increase* and *decrease*) that the task manager class uses to request a task to increase or decrease processing demands. The developer is responsible for defining the actual implementation of these methods in derived classes.

The task class also defines an important method (*sync*) that is responsible for synchronizing the task using whatever synchronization object is available (e.g., barriers, mutexes, or semaphores). We provide a default implementation of this method, using a barrier as the synchronization object. However, we consider that the tasks are independently from each one other, which means that each task runs alone and the synchronization is employed to guarantee the visualization. Others synchronization objects can be implemented.

The Task Manager is also responsible for distributing the threads among available multi-core processors and distributing tasks among the threads. We adopt a FIFO policy to schedule game tasks. Although FIFO is a simple scheduling policy, it works well in games because all tasks must be computed in a discrete time step ($1/30$ or $1/60$). Although more sophisticated scheduling policies could be implemented, they could require more processor time, which might affect the game loop adversely. Furthermore, desktop computers are not dedicated hardware, they have several processes run concurrently, sharing the CPU, even if CPU multi-core, with the game tasks and other processes. We have in mind that CPU is shared between the Task Manager (scheduler) and the game tasks.

At last, we used C++ with pThread and CUDA libraries in order to program in parallel model. The former works on CPUs multi-core, while the latter is employed on GPUs.

IV. Case Studies

This section describes two case studies we have performed to test the proposed architecture. To compare the influence of applying the tardiness strategy, each case study provides a test that adjusts task operation according to the tardiness metric ("*tardiness-applied-test*") and another test that does not adjust task operation ("*tardiness-not-applied* test"). Both tests calculate tardiness; the first one applies the adaptive approach while the second one uses tardiness to understand task behaviour (e.g., if a task takes too long to process or if it finishes early), using these data to compare results with the adaptive approach.

The first case study uses an interruptible game task and dynamic thread management on a multi-core CPU scenario. This case study aims at investigating if the proposed architecture is able to adjust threads dynamically on demand, when applying the tardiness strategy.

The second case study is a synthetic non-interruptible game task that represents an accurate physics simulation with high processing demands. Generally, digital games do not use these kinds of tasks due to the required high computing demands. However, we wanted to evaluate our architecture with a heavy non-interruptible task to understand how these kinds of tasks would affect a game simulation. This second case study solves a physics simulation on a multi-GPU scenario (a non-interruptible task). The physics simulation simulates an explosion (sound wave propagation) that travels through an environment with large obstacles.

The tests in these case studies start computing tardiness only after the first 100 frames to avoid the application transient state. The application transient state occurs when the application starts running, due to events related to application start up (such as library initialization, memory allocation, and thread creation, among others). These events may impact tardiness calculation artificially, so the architecture waits some time before calculating tardiness.

The case studies use $1/60$ as the target time step ($Time_{Programmed}$ parameter in Eq. 2). In each case study, the test applications record frames per second (FPS), game task elapsed times (ET), and tardiness ($Tardiness$) values.

The test platform is a desktop machine using an Intel i7 (with four 3.60Hz physical cores), 8GB RAM, PCI-express 3.0 bus (1.8 GHz memory access), and two NVidia 640 GTX GPUs, running the Ubuntu 12.04 64 bits operating system. All tests were implemented using C++, OpenGL, and GLUT. The remaining of this section discusses the case studies and presents the results.

A. Case Study 1: A* Pathfinding

The first case study implements an A* pathfinding algorithm, using a 2D maze scenario similar to scenarios found in earlier Pac-Man games. In this case study, several agents (i.e., the ghosts) calculate the path (using A*) towards the main character (i.e., Pac-Man) to try to capture it. Each agent is controlled by a different thread. Figure 5 illustrates the test maze, where walls are represented in red colour and corridors in black. We wanted to use a test maze much larger than a typical Pac-Man scenario, so we chose to design a test scenario of 128×64 tiles in size.

The starting position of Pac-Man is at the upper-right corner, while the ghosts start at the bottom-left. Figure 5 depicts the characters positions on the scene. We chose these initial positions to characterize the worst case for the A* algorithm to process.

This case study was designed to demonstrate the flexibility of the proposed architecture. The tests (tardiness-applied and *tardiness-not-applied*) implement a scalable A* pathfinding

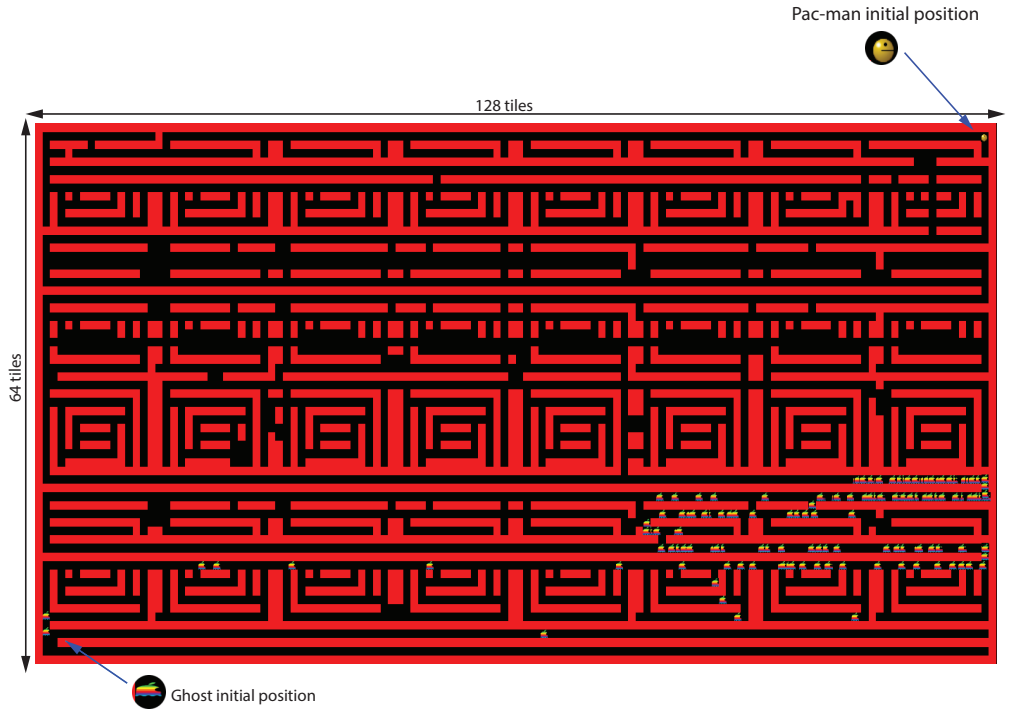

Fig. 5: *2D maze scenario.*

algorithm, which increases or reduces the number of agents (ghosts) according to the host hardware performance.

Both kinds of tests (tardiness-applied and *tardiness-not-applied*) have initial conditions regarding the starting number of threads. We ran 12 instances of each kind of test, using different values for the starting number of CPU threads: $(1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 640,$ and $768)$. As A* pathfinding algorithm requires heavy memory access due to the characteristics of algorithm (more data access than CPU computation), this algorithm was implemented on CPU. Moreover, we considered that each ghost computes its own A* pathfinding algorithm independently of the others. Thus, each thread controls just one ghost and the total of threads is scheduled by O.S. among the CPU cores.

We adopted **pthread** library and semaphore as synchronized object in order to guarantee the consistency of ghosts position their visualization.

For *tardiness-applied-test*s, we expect that these instances converge to the ideal number of threads when applying tardiness, given a target time step of $1/60$. The *tardiness-not-applied* tests do not change the number of threads dynamically, using the number of ghosts defined before the test started.

B. Results

In this section, Figures 6, 7, and 8 illustrate the test results in green (*tardiness-applied-test*) and in red (*tardiness-not-applied* test).

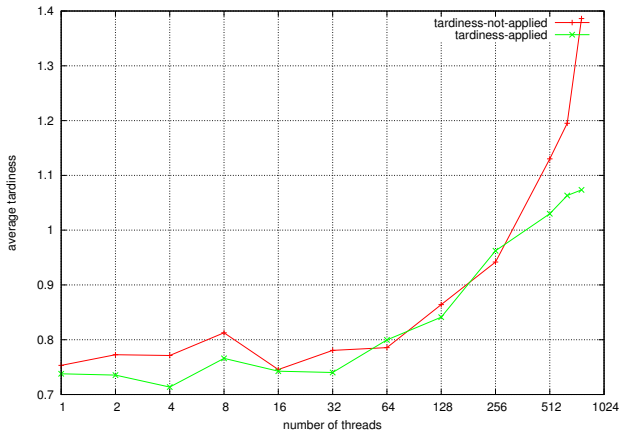

Fig. 6: *Average tardiness.*

Figure 6 illustrates average tardiness behaviour per frame while varying the number of threads (ghosts) running in the system. The tests record the average tardiness every 100 frames. The tardiness calculations consider all tasks that run in the tests, which includes the A* algorithm and rendering.

Considering the *tardiness-applied-test*, Figure 6 illustrates that average tardiness does not vary significantly while the number of threads range from 1 to 32. We suspect that this situation occurred due to rendering being the task that most contributes

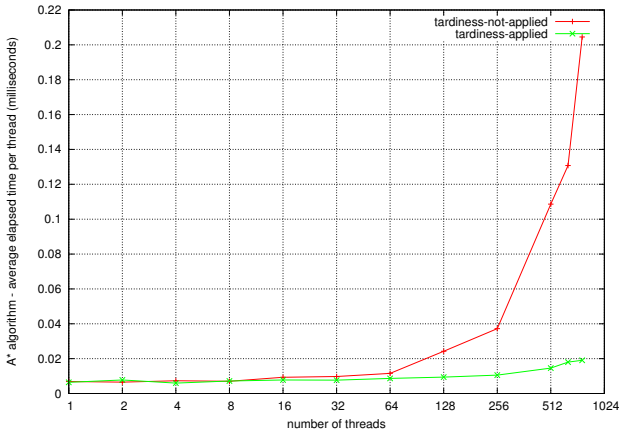

Fig. 7: *A* pathfinding: average elapsed time per thread (milliseconds).*

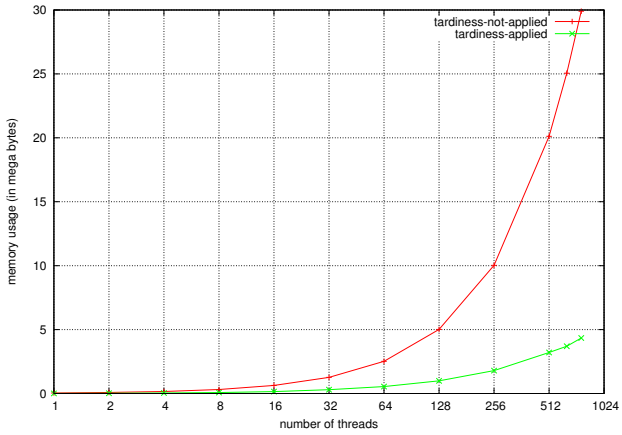

Fig. 8: *Memory usage (megabytes).*

to tardiness when the number of threads ranges from 1 to 32. This situation requires further investigations as our tests do not measure total rendering time. When the number of threads is greater than 64, the influence of all threads on tardiness start to have more significant impact.

Figure 7 illustrates average elapsed times when processing a frame, for each thread (A* algorithm). The *tardiness-applied-test* spends much less time to finish than the *tardiness-not-applied* tests when the number of threads is greater than 64. When the *tardiness-not-applied* test runs with more than 256 threads, the test always exceeds the target time step $(1/60)$, while the *tardiness-applied-test* is able to keep total processing time below the target time step when using up to 768 threads. All *tardiness-applied-test*s converge to using 256 as the ideal number of threads, when the tardiness value is approximately 1.

Figure 8 depicts memory usage in both tests. The *tardiness-applied-test* used less memory than the *tardiness-not-applied* test in overall. This difference starts to increase significantly from 32 threads. This is an interesting result, although we regard it as side-effect contribution because our research was not concerned with reducing memory usage while applying tardiness control. Also, we cannot guarantee that this result applies to other kinds of game tasks.

C. Case Study 2: Shock-wave Explosion

The second case study is a synthetic game task, it corresponds to a shock wave simulation that models how a shock wave (originated from an explosion) travels through an environment with large obstacles, such as a city block with tall buildings. The application used the resulting amplitude field to render the propagation of a shock-wave-like effect at each frame step.

This case study models an outdoor scene that is represented by a lattice with cells, using a physics simulation to model the shock wave propagation. Figure 9 displays the test scene geometry and the evolution of the shock-wave propagation effect in time. The wave propagation starts in (A) and ends in (D). The buildings interfere in the wave propagation. Parts (A) and (B) omit buildings to help in visualizing wave reflection. Parts (C) and (D) present the complete scene.

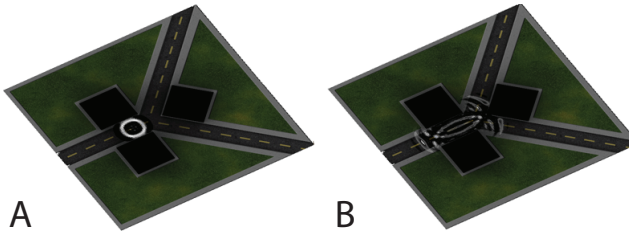

Fig. 9: *Test scene geometry and the wave propagation effect evolving through time (from A to B). Parts (A) and (B) omit buildings to help in visualizing wave reection.*

The shock wave propagation is implemented by finite difference methods (FDM) on the GPU. The FDM requires two parameters: $\Delta h = 1.0$ meter and $\Delta t = 0.0033$ seconds. The tests in this case study starts with a domain size of 128×128 points. The largest domain size that the tests use is 4096×4096. The domain size describes how many cells exist in a lattice. Domain sizes larger than 4096×4096 are infeasible to process using the test hardware. The number of threads is equal to the number of cells. Hence, each thread computes one cell. We defined blocks with 32×32 threads, composing total of $1,024$ threads (limited by hardware - NVidia 640 GTX). Besides, shared memory size is based on the number of threads plus a region called buffer border whose size is two times the neighborhood size, corresponding to the 2D stencils FDM size, i.e., considering 32 bits float point and block size of 32×32, then we allocated approximately 4 Kbytes of shared memory, as we proposed in [33].

The *tardiness-applied-test* increases the domain size when tardiness is less than 1 and decreases the domain size when tardiness is above 1. This test applies the following strategy to

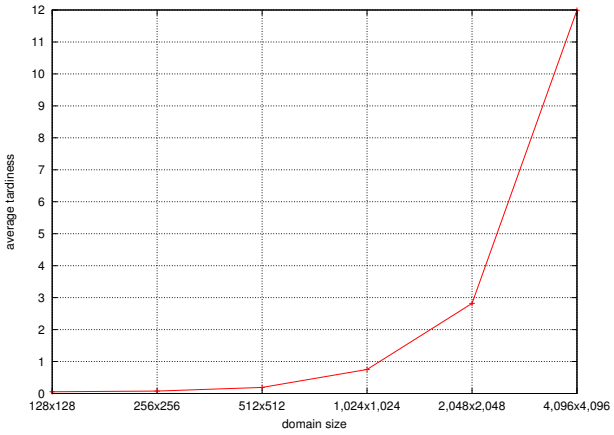

Fig. 10: *Average physics simulation tardiness.*

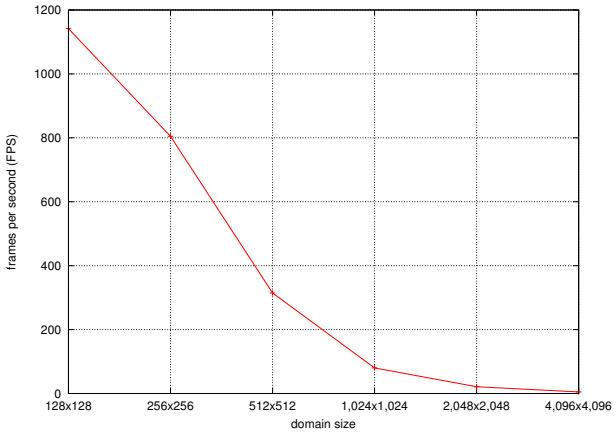

Fig. 11: *Physics simulation FPS.*

change the domain size: when it needs to increase the domain size, the doubles it (e.g., from 128×128 to 256×256). When the test needs to reduce the domain size, it halves it (e.g., from 512×512 to 256×256). The *tardiness-not-applied* test does not change the domain size dynamically, running with a fixed domain size defined before the test starts. Domain sizes larger than $4,096 \times 4,096$ are infeasible to process using the test hardware.

In this approach, the CPU invokes the simulation on GPU, transferring the necessary data to simulation and processing to GPU. The NVidia Graphic Card driver instances and schedules threads among GPU processors and CUDA cores. The developer only defines the number of blocks, threads per blocks and the codes (kernels) which are executed by GPU. However, the CPU can abort the GPU execution or defines a smaller domain, so that GPU can execute in fast way.

D. Results

Initially, we ran several rounds of the *tardiness-not-applied* test, using different domain sizes ranging from 128×128 to $4,096 \times 4,096$. In these rounds we measured average tardiness (AT), average frames per second (FPS), average elapsed time per frame (ET), and the amount of memory allocated (Mem). Table I presents these quantitative results.

Domain	A. T.	E. T.	FPS	Memory
128x128	0.053	0.876	1141.711	0.063
256x256	0.075	1.243	804.687	0.250
512x512	0.191	3.181	314.374	1.000
1,024x1,024	0.751	12.509	79.941	4.000
2,048x2,048	2.819	46.983	21.284	16.000
4,096x4,096	11.991	199.847	5.004	64.000

TABLE I: *Physics simulation performance.*

Considering the test hardware, the most suitable domain size to use was $1,024 \times 1,024$, when tardiness averages approximately 0.751. For larger domains, tardiness is greater than 1. Figure 10 illustrates average tardiness according to domain size.

Figure 11 illustrates average FPS in *tardiness-not-applied* tests. When the domain size is larger than $1,024 \times 1,024$, the average FPS is greater than 79.941. On the other hand, when using domain sizes larger or equal than $2,048 \times 2,048$, the average FPS rate drops significantly (Table I presents these values).

Since this task is non-interruptible, memory usage does not vary while the test runs. The reason is that the physics simulation allocates all required memory at once when it starts according to the defined domain size. Figure 12 illustrates memory usage when considering different domain sizes.

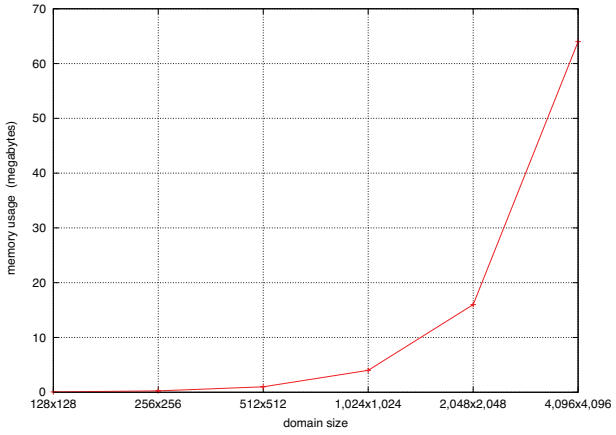

Fig. 12: *Memory usage.*

The *tardiness-applied-test* starts running using the domain size 128×128 and changes this value until it reaches $2,048 \times 2,048$, when it discovers that this domain size is infeasible because tardiness is greater than 1. The test then chooses the last suitable value $(1,024 \times 1,024)$ for the domain size and keeps on using while tardiness ranges from 0.8 to 1. The test uses this range to avoid constant domain size switching (e.g., from $1,024 \times 1,024$ to $2,048 \times 2,048$, back and forth). On the other hand, when tardiness is below 0.8, the test decreases the domain size to the previous suitable value (e.g., 512×512).

When comparing the *tardiness-applied-test* and the *tardiness-not-applied* tests, there is no difference regarding the quantitative results (average tardiness, average FPS, average elapsed time, and memory usage). The reason is because the physics simulation in these tests is non-interruptible, which means that the task needs to run until completion in a single time step.

The main difference between the *tardiness-applied-test* and *tardiness-not-applied* tests is that the *tardiness-applied-test* is able to adjust domain size dynamically according to the current hardware conditions (e.g., free processing time available, processing capacity), which may improve task results. On the other hand, the *tardiness-not-applied* test uses fixed domain size, which may result in poor performance if the domain size is not adequate to the current hardware conditions.

V. Conclusion and future works

This paper proposed an adaptive game loop architecture based on tardiness control to adapt task operation according to the current hardware load conditions at a given moment. Furthermore, our adaptive game loop is able to create game task list scheduling, considering multi-core processors. Despite creating the list scheduling correctly based on GA Algorithm, we did not compare to analytic solution and, therefore, we do not know how far the list scheduling is from the best arrangement. So that it is NP-hard problem.

Applying tardiness control in games is a new approach that may help game developers in designing tasks that run adequately in a myriad of hardware configurations and running conditions. To the best of our knowledge, the research we report in this paper represents the first attempt to apply tardiness policies in game architectures.

Firstly, the architecture proposed in this paper uses a simulation model based on time steps of fixed size in order to create deterministic simulations. In other words, the architecture is able to reproduce a simulation given the same set of input data. This target time step value should be a suitable value so that the game simulation remains interactive. In practice, this value ranges from 1/30 to 1/60 seconds.

However, when task processing time exceeds the target time step, the simulation starts to run slower than real-time, which violates the real-time requirements that games have. This situation may arise for several reasons, such as running hardware not meeting the minimum task run-time requirements and temporary high processing loads that may occur due to game events (e.g., the game generated an explosion that sparked several particle systems). When developing a game, it is difficult to foresee when these kinds of situation will happen due to the variety of computer hardware configurations and unpredictability of game events.

Tardiness control seems to be a promising approach to handle these kinds of situations, as we understood by developing two prototypes using the proposed architecture. The proposed architecture monitors the simulation running times to learn about the behavior of total task processing time when compared to the target time step size (in percentage). When calculating tardiness, the result means that 1) task processing time exceeds the target time step; 2) task processing time is below the target time step; and 3) task processing time equals the target time step (the optimal case, which is hard to reach in practice). The proposed architecture uses this percentage value to request tasks to adjust their operation dynamically. Using tardiness also makes it possible to use extra time available to improve the complexity and results of game tasks automatically, which may improve the game experience. We understood that applying tardiness in games seems to be a good measure to keep the simulation bound to the real-time requirement (which we represent by keeping task processing time bounded by the target time step).

In this paper, we discussed applying tardiness control for interruptible and non-interruptible tasks. Interruptible tasks need to guarantee their continuity in other words, these tasks must be capable of resuming their work after being interrupted. This is a crucial requirement in our approach those tasks must not reset after being interrupted. The prototypes demonstrated that adapting tasks dynamically using tardiness is a feasible and promising approach. However, while doing the research we report in this paper, we noticed some aspects that require further investigation.

Currently, the proposed architecture calculates tardiness and requests all tasks to change based on this value. For example, if the simulation has three tasks (A, B, and C) and one of them (C) is heavily responsible for exceeding the target time step, the current approach penalizes all three tasks. The next step in applying tardiness policies would be applying a weighted approach when requesting tasks to reduce their processing requirements. In other words, in this example the task that contributes the most to total processing time (C) is the one that most needs to reduce processing requirements.

Another important issue relates to non-interruptible task processing. Currently, the proposed architecture needs to run a non-interruptible task entirely before calculating tardiness. In this case, the real-time requirement and interactivity may be at jeopardy, as these tasks might take too long to process. Hence, a future work is proposing more policies to handle tardiness calculations when considering non-interruptible tasks. Finally, another important future work is creating scheduling strategies to maximize game task processing considering multi-core CPUS and programmable GPUs.

In future work, we will investigate GA algorithm to create game list scheduling. Specially, in related to real game tasks.

Acknowledgment

The authors gratefully acknowledge CNPq, CAPES, FINEP and FAPERJ for the financial support of this work.

REFERENCES

[1] D. S.-C. Dalmau, *Core techniques and algorithms in game programming*. New Riders, 2004.

[2] A. LaMothe, *Tricks of the Windows game programming gurus*. Sams Publishing, 2002.

[3] M. Claypool, K. Claypool, and F. Damaa, "The effects of frame rate and resolution on users playing first person shooter games," in *Electronic Imaging 2006*. International Society for Optics and Photonics, 2006, pp. 607 101–607 101.

[4] D. Andreev, "Real-time frame rate up-conversion for video games: or how to get from 30 to 60 fps for free," in *ACM SIGGRAPH 2010 Talks*. ACM, 2010, p. 16.

[5] K. T. Claypool and M. Claypool, "On frame rate and player performance in first person shooter games," *Multimedia systems*, vol. 13, no. 1, pp. 3–17, 2007.

[6] Microsoft, "Xbox one is built for the future." Avalible at: howpublished = Available at http://www.gamasutra.com/view/news/295800/Inside_the_next_Xbox_Project_Scorpio_and_its_brandnew_dev_kit.php, 2015, last accessed 14 Abril 2017.

[7] engadget, "Xbox one hardware and specs: 8-core cpu, 8gb ram, 500gb hard drive and more," Avalible at: http://www.engadget.com/2013/05/21/xbox-one-hardware-and-specs/, 2015, last accessed 14 Abril 2017.

[8] Sony, "Playstation 4 specifications," Avalible at: https://www.playstation.com/en-gb/explore/ps4/tech-specs/, 2015, last accessed 14 Abril 2017.

[9] G. Singh, K. Kaur, and A. Chhabra, "Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system," *International Journal of Computer Science and Security (IJCSS)*, vol. 4, no. 2.

[10] P.-G. Taranti, C. J. P. de Lucena, and R. Choren, "An architecture to tame simulation time tardiness in ads," in *Proceedings of the 2011 Workshop on Agent-Directed Simulation*, ser. ADS '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 29–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048355.2048359

[11] M. H. Kim and Y.-D. Kim, "Simulation-based real-time scheduling in a flexible manufacturing system," *Journal of manufacturing Systems*, vol. 13, no. 2, pp. 85–93, 1994.

[12] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[13] T. Morton, *Heuristic scheduling systems: with applications to production systems and project management*. John Wiley & Sons, 1993, vol. 3.

[14] C.-F. Liaw, Y.-K. Lin, C.-Y. Cheng, and M. Chen, "Scheduling unrelated parallel machines to minimize total weighted tardiness," *Computers & Operations Research*, vol. 30, no. 12, pp. 1777 – 1789, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0305054802001053

[15] J. P. de CM Nogueira, J. E. C. Arroyo, H. M. M. Villadiego, and L. B. Gonçalves, "Hybrid grasp heuristics to solve an unrelated parallel machine scheduling problem with earliness and tardiness penalties," *Electronic Notes in Theoretical Computer Science*, vol. 302, pp. 53–72, 2014.

[16] C. Koulamas, "The total tardiness problem: Review and extensions," *Operations Research*, vol. 42, no. 6, pp. 1025–1041, 1994. [Online]. Available: http://dx.doi.org/10.1287/opre.42.6.1025

[17] T. Sen, J. M. Sulek, and P. Dileepan, "Static scheduling research to minimize weighted and unweighted tardiness: a state-of-the-art survey," *International Journal of Production Economics*, vol. 83, no. 1, pp. 1–12, 2003.

[18] L. Valente, A. Conci, and B. Feijó, "Real time game loop models for single-player computer games," in *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 2005, pp. 89–99.

[19] D. S. C. Dalmau, *Core Techniques and Algorithms in Game Programming*. New Riders Publishing, 2003.

[20] H. Gabb and A. Lake, "Threading 3d game engine basics," Available at http://www.gamasutra.com/view/feature/130873/threading_3d_game_engine_basics.php, 2005, last accessed 14 Abril 2017.

[21] V. Mönkkönen, "Multithreaded game engine architectures," Available at http://www.gamasutra.com/view/feature/130247/multithreaded_game_engine_.php , 2006, last accessed 14 Abril 2017.

[22] M. Zamith, E. Clua, A. Conci, and A. Montenegro, "Parallel processing between gpu and cpu: Concepts in a game architecture," in *Computer Graphics, Imaging and Visualisation, 2007. CGIV '07*, 2007, pp. 115–120.

[23] M. P. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, R. C. P. Leal-Toledo, P. A. Pagliosa, L. Valente, and B. Feijó, "A game loop architecture for the gpu used as a math coprocessor in real-time applications," *Comput. Entertain.*, vol. 6, no. 3, pp. 1–19, 2008.

[24] M. Joselli, M. Zamith, E. Clua, R. Leal-Toledo, A. Montenegro, L. Valente, B. Feijo, and P. Pagliosa, "An architeture with automatic load balancing for real-time simulation and visualization systems," *JCIS - Journal of Computational Interdisciplinary Sciences*, pp. 207–224, 2010.

[25] A. E. Rhalibi, S. Costa, and D. England, "Game engineering for a multiprocessor architecture," in *DIGRA Conf.*, 2005.

[26] W. AlBahnassi, S. Mudur, and D. Goswami, "A design pattern for parallel programming of games," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, Jun. 2012, pp. 1007–1014.

[27] M. J. Best, A. Fedorova, R. Dickie, A. Tagliasacchi, A. Couture-Beil, C. Mustard, S. Mottishaw, A. Brown, Z. F. Huang, X. Xu, N. Ghazali, and A. Brownsword, "Searching for concurrent design patterns in video games," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin Heidelberg, Jan. 2009, no. 5704, pp. 912–923. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-03869-3_84

[28] Y.-C. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputing'92., Proceedings*. IEEE, 1992, pp. 512–521.

[29] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[30] S. Russell, P. Norvig, and A. Intelligence, "A modern approach," *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, vol. 25, p. 27, 1995.

[31] R. C. Corrêa, A. Ferreira, and P. Rebreyend, "Scheduling multiprocessor tasks with genetic algorithms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 8, pp. 825–837, 1999.

[32] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 2, pp. 113–120, 1994.

[33] M. Zamith, E. Passos, D. Brandao, A. Montenegro, E. Clua, M. Kischinhevsky, and R. C. Leal-Toledo, "Sound wave propagation applied in games," in *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*. IEEE, 2010, pp. 211–219.