Game loop model properties and characteristics on multi-core CPU and GPU games

*Marcelo Zamith[1] Luis Valente[2] Bruno Feijo[3] Esteban Clua[2]

Universidade Federal Rural do Rio de Janeiro, Computer Science Department, Multidisciplinary Institute, Brazil[1]

Universidade Federal Fluminense, Institute of Computing, Brazil[2]

Pontifícia Universidade Católica do Rio de Janeiro, Department of Informatics,VisionLab/PUC-Rio, Brazil[3]

ABSTRACT

From a software architecture point of view, digital games are interactive systems that create virtual environments with the illusion that everything is happening at the same time (i.e. in real-time). This illusion is carried out through "game loop models" (or architectures). Although this illusion is a key feature in these applications, the patterns and requirements that lead to its implementation have been taken for granted in the literature. Even though there are works about specific game loop model architectures, the literature lacks solid foundations about underlying game loop principles and requirements. This paper contributes to this area by presenting and classifying these issues, considering homogeneous and heterogeneous hardware processing requirements (e.g. multi-core CPUs and GPUs) that are available for games and interactive applications

Keywords: game architectures, game loop, game task flow

1 INTRODUCTION

Digital games create interactive and dynamic virtual environments where events seem to happen according to our everyday perception of time – for example, continuous motion and instant input feedback. This effect is an illusion (i.e. the game illusion) that the game application creates by presenting media (e.g. images, animations, audio, haptics feedback) in continuous cycles (i.e. frames) at a high frequency. This presentation process follows several other tasks (e.g. physics simulation, AI, game logic) that a game processes to compute the current "game state" – a static snapshot of data and variables that describes the game at a given moment. The game illusion is a key component of the experience that a game provides, which also involves aspects such as fun, immersion, learning, flow, engagement, and interactivity.

From a software architecture point of view, the model that computes the current game state can be regarded as a "game loop" – a structure that defines the ordering of task computations. Although game loops are of central importance in game development, there is scarce literature about game loop fundamentals and components (e.g. [1]–[3]). There are few academic works that describe specific game loop architectures, but usually this subject is taken for granted in game development. In this paper, we propose a set of concepts describing game loop fundamentals, which we derived from studying several works related to this subject. We also present a concurrency view about game loop models found in the literature considering the framework we describe in this paper.

In Section 2 we discuss game loop fundamentals considering discrete simulations, game tasks, real-time concepts, and interactivity. In Section 3 we present general game loop properties. Section 4 presents task flow organizations (i.e. game loop models) considering concurrency aspects – although game loops have been around for a long time, the availability of multi-core CPUs and GPUs for general processing require attention to parallel programming issues in games. Finally, Section 5 presents conclusions and future works.

2 GAME LOOP FUNDAMENTALS

This section explores the main concepts related to game loops: discrete simulations, game tasks, time flow structures, and interactivity.

2.1 Discrete simulations

The computer game simulation usually advances in time using discrete a time-stepped mechanism, although there are event-based approaches [2], which are not the focus of this paper.

In discrete simulations, the game application creates a virtual environment by computing discrete game states sequentially across time. In this context, there are three key concepts related to time: wall-clock time, simulation time, and real-time. *Wall-clock time* corresponds to the passage of time as perceived by human uses, being the time we see on everyday clocks. *Simulation time* is the time represented in a simulation (or game) timeline. In simulations based on time steps, the timeline corresponds to a collection of contiguous discrete time spots. Figure 1 illustrates a game timeline based on time steps, where all time steps have the same size. In practice, a game timeline may also have time steps of different sizes, according to the time structure applied. Section 2.3 explores these issues.

Figure 1: A sample timeline as a series of sequential time steps

The simulation may advance time in the simulation timeline according to various paces. The simulation runs in *real-time* when time advances occur according to wall-clock time pace.

In time-step based simulations, the game calculates a game state (or frame) for the current time step and advances the simulation to the next time step to compute the next game state. We define as *simulation sample rate* as the number of time steps that exist in a second of simulation. We define as *presentation sample rate* the number of game states that the game presents (as images) per second. We use the term "FPS" (frames per second) as the unit to express these sample rates. There are game loop approaches where the simulation sample rate and the presentation sample rate are the same, while there are approaches where these values are different. Section 2.3 discusses time flow structures that lead to these approaches.

[1]email: mzamith@ufrrj.br

The simulation that a game creates does not need to be highly accurate (e.g. physics simulations). The simulated processes need to have a sample rate that produces "good enough" results – results that represent *sufficient fidelity* (Section 2.2.1). If the simulation runs a process more frequently than required, the simulated process will be *oversampled* and the game will waste computational resources. In contrast, "analytical simulation" is another simulation structure that has different goals from game based simulations. For example, analytical simulations usually are concerned with modeling highly accurate processes to produce data for quantitative analysis [4]. So, analytical simulations need to be very accurate and usually advance time as fast as possible, not including human interaction [4]. In games, human interaction is an intrinsic component and the simulation runs in real-time.

2.2 Game tasks

Digital games manage a variety of heterogeneous subsystems such as rendering, input acquisition, audio playing, artificial intelligence, network management, character animation, graphics user interface management, resource management (e.g. I/O systems and media loading), scripting, and physics simulations. We define as a "game task" any processing or computation that a game must perform in these subsystems. These tasks can be grouped into three general categories: input data acquisition, game state computing (i.e. simulation), and presentation. Figure 2 illustrates the most basic game loop model, which runs tasks sequentially in a loop. This model is referred in this survey as the "simple coupled model" [1].

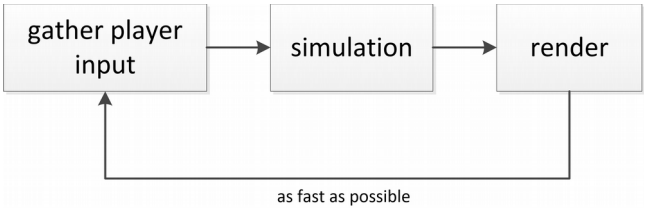

Figure 2. Simple coupled model

The input data acquisition category corresponds to tasks that receive input from various devices. There are a multitude of input devices that a game should be able to handle, such as keyboards, joysticks, voice input, location sensors (e.g. GPS), motion sensors (e.g. Kinect), orientation sensors (e.g. accelerometer, gyroscope), and touch screens.

The simulation category corresponds to tasks that compute the current game state, by computing the game logic (i.e. game rules) and computing several processes such as physics simulations and artificial intelligence behavior.

The presentation category corresponds to tasks that present the current game state to the players though several outputs, such as images, audio, video, animation, and haptics feedback.

In a running simulation, the set of active game tasks is dynamic: player interaction with the game world and the enforcement of game rules generate events in intermittent fashion. For example, a player character may collide with an object, generating an explosion event. The explosion event may generate particles, which require a particle system processing task and audio playing to provide feedback to the player about the explosion. The game runs these tasks, and when they are finished, the game removes them from the set of active tasks.

2.2.1 Task properties

We identify and define three properties for game tasks: *nature*, *sample rate*, and *deadline*.

The *nature* of a task relates to the kind of computation that the task needs to perform. For example, game tasks may perform functional calculations, state-based calculations, and data retrieval [5]. Functional calculations transform or generate data (e.g. solving mathematics calculations, computing physics simulations, and manipulating data structures). State-based calculations refer to operations that need to maintain and manipulate states across time (e.g. game logic state, animations, and AI state machines). Data retrieval tasks include loading media resources that are necessary for game state presentation.

The *sample rate* property refers to the number of times a task is processed per time unit (e.g. usually seconds). This property is usually expressed in FPS, although "frame" is a concept originally used to refer to the resulting image that a game displays on the screen at the end of game loop processing. Game tasks have a minimal sample rate that is required for a task to produce results of *sufficient fidelity* ("good enough results"). Different tasks have different sample rate requirements. For example, a hypothetical physics simulation sampled 30 times per second may yield data good enough to produce a smooth animation. If this sample rate is higher than 30 FPS, the physics simulation will be more accurate but the player may not be able to notice any differences when compared to less accurate version. In this case, the physics simulation will be *oversampled*. On the other hand, if the sample rate is not large enough to provide results of sufficient fidelity, the simulated process will be *undersampled*. Determining the ideal (or optimal) sample rate for game tasks is a central problem in game development – different tasks have different sample rate requirements and hardware configurations vary.

The *deadline* property is directly related to the sample rate property, being the time required for a task to finish processing in the current time step: if a task sample rate is 30 FPS, the task must be finished in 33ms. In practice, the actual deadline that a task has will be much lower, subject to the workload and the rendering target sample rate. Usually, the target sample rate for rendering is 60 FPS (i.e. an approximate deadline of 16ms).

2.2.2 Task dependencies

Task dependency means that a task depends on some external data or event before running. El Rhalibi et al. [6] identified some categories of task dependencies such as flow dependencies, antidependencies, output dependencies, I/O dependencies, and control dependencies. Task dependencies in games generally takes form as data dependencies, which means that a task requires data produced by another task in order to proceed. In other words, a task "A" must wait for some other tasks (e.g. "B" and "C") to finish before receiving the required data. Otherwise, task "A" will operate with incorrect data leading to incorrect and inconsistent results. Some examples of task dependencies in games are:

1. The game is unable to evaluate game rules before receiving player input;
2. Non-player characters are unable to move before AI processing is finished;
3. Rendering requires an entirely computed game state.

To ensure simulation consistency, the simulation must consider all task dependencies and ensure to run tasks in the correct order. In Figure 2 (the "simple coupled model"), the arrows express task dependencies. The game loop starts with reading player input, feeding this information to the simulation stage, which computes the current game state. Next, the rendering presents the game state to the player, and the loop is restarted.

2.2.3 Game tasks and parallelism

In game development, parallel computing usually explore functional parallelism and/or data parallelism. In functional (or task) parallelism there are several concurrent threads dedicated to process a specific task or subsystem (such as AI and rendering), or a group of tasks that have high interdependencies. The threads may run asynchronously and the rendering task presents the latest game state that was completely computed. In data parallelism, the system distributes data in separate parallel nodes, where each node runs the same tasks or sequence of tasks. Data parallelism is useful to process data that has little or no interdependencies, as these would require synchronization points that could diminish performance gains.

The performance gains of paralleling tasks can be estimated using the Amdhal's law [7] and Gustafson's law [8]. These laws state that performance gains due to parallelizing tasks are proportional to the number of parallel processors and limited by the number of serial parts. Amdahl's law considers that the entire problem has fixed size and Gustafson's law considers that the amount of parallel work varies linearly according to the number of parallel processors.

Game tasks usually have several interdependencies that must be addressed to ensure consistency. In parallel programming environments this issue requires task synchronization, which may be applied through semaphores, mutexes, and barriers.

An important issue related to using parallel programming in games regards the integration with third-party libraries and tools, because they may have been developed without considering parallel programming and multi-thread environments. These tools might not be thread-safe and may apply internal multi-thread approaches that cannot be controlled or accessed by the game developer.

Another important concept regarding parallelism is heterogeneous processing, which means processing tasks in a system composed of processors with different architectures. Heterogeneous processing in games is becoming more common as GPUs are being used for non-rendering tasks. GPUs have been designed to solve problems that are modeled as stream-based processes with massive mathematical calculations. The intense GPU computing capability enables games to process tasks such as linear algebra, artificial intelligence, and physics simulations.

When designing game tasks to run in heterogeneous environments, it is important to consider the strengths and weakness of each kind of processor. For example, multi-core CPUs (MIMD architecture) can manage hundreds of threads in different set of data. On the other hand, GPUs (SIMD architecture) are able to manage thousands of threads, but these threads operate on the same set of data. In this regard, data locality is crucial for GPGPU performance due to the SIMD architecture of GPUs [9]. Another issue in heterogeneous environments is the cost related to communication among processors that have different architectures. For example, in GPGPU applications the GPUs are not able to process data while CPUs are accessing the main RAM. Also, memory reading access in GPUs is an operations that presents high latency in this type of hardware [10].

2.3 Real-time and time flow structure

A game runs in real-time when the game simulation advances the game timeline according to wall-clock time pace. A computer game has real-time requirements because if the game fails to advance the game simulation according to wall-clock time pace, the user experience will be severely impaired, thus breaking the "game illusion". This failure can occur if (for example):

1. Being unable to process all tasks before the current time step expires (i.e. the deadline);
2. Processing all tasks too fast and not waiting for the next time step. For example, if a game uses 33ms as the time step size and the all task processing takes 10ms to complete, the game needs to wait for 23ms before executing the next time step. Otherwise, the game runs faster than wall-clock time, as in a movie played fast-forward.

Computer games may be considered as "soft-real time applications", which differs from a "hard real-time system". The latter has stricter time requirements because if these systems fail, severe consequences may occur. Soft real-time systems may have more tolerance to time delays and loss of wall-clock time pace. For example, games may be able to recover from task delays and keep the simulation running in real-time. However, if delays are too high the game experience will be severely compromised.

Game development literature commonly regards that game performance should stay in the 30-60 FPS range to keep the game experience smooth, which includes the game presenting smooth motion and responsive input. Game developers usually target a minimum presentation sample rate of 30 FPS as a "safe starting point" to avoid jerkiness, unresponsive input, general slowness, and other undesired side-effects of low frame rates in games. Another problem related to game frame rates is temporal aliasing. Temporal aliasing appears as jittery and unnatural motion, which occur when the simulation and/or presentation sample rates are too low to represent fast moving objects adequately. Solutions to handle temporal aliasing include applying motion blur techniques to the resulting images [11] and fine-tuning simulation and presentation frame rates.

Depending on the time flow structure that the game simulation uses, we classify the simulation into two groups: coupled and uncoupled simulations. Sections 2.3.1 and 2.3.2 discuss these two groups.

2.3.1 Coupled time flow structures

In coupled simulations, the actual simulation sample rate is variable and directly dependent on the presentation sample rate. This means that the simulation time step size depends on the host hardware and current workload, and thus may vary while the game is running. Consequently, players will perceive the game as running faster than wall-clock time in more powerful machines, and slower than wall-clock time in less powerful machines, which means lack of *simulation uniformity*. If more computational resources are available these approaches are not able to improve the quality of the game experience, which means lack of *adaptability*. Consequently, this leads to *wasting computational resources and power*, which are aspects that become crucially important when the game runs on mobile devices equipped with batteries that have limited autonomy. As a result, coupled simulations may fail to satisfy real-time requirements in many situations.

In this scenario, if game tasks start to take too long to complete, the FPS rate will drop with very noticeable undesired effects . This may lead to an effect known as "frame rate spike" [12], which happens when the game frame rate varies abruptly too fast, undermining the game experience. Players may perceive this performance degradation as unresponsive input and sluggish animations, for example.

Another important issue in these approaches is the *lack of simulation control*, as explicit simulation time does not exist. This means that the simulation *is not repeatable* – it is not possible to reproduce the simulation given the same input data (i.e. simulation becomes non-deterministic). As all tasks have the same sample rate, this situation possibly leads to oversampled tasks. Lack of

simulation control hinders the implementation of certain game features (as gameplay replays and networking synchronization) and makes game development harder (e.g. poor debugging capabilities)

These problems arise mainly because the simulation is coupled with the frame rate – the frequency of running the simulation is regulated by the frame rate. In the past, using these approaches were common because hardware was more uniform (e.g. games for video game consoles) and hardware had much limited capacities. Old arcade game machines are a nice example of such kind of hardware, as frequently the game machines were built specifically for a particular game.

2.3.2 Uncoupled time flow structures

Uncoupled simulations aim at satisfying real-time requirements by having the simulation and presentation running with different (and independent) sample rates. Common approaches are: 1) scale calculations using elapsed time to have the simulation run in real-time; and 2). Defining a fixed sample rate for the simulation, which gets rid of all elapsed time scaling.

In the first approach, the simulation measures the time elapsed of a complete processing cycle (i.e. input-simulation-presentation) and feeds this value to the next game state computation, exploiting *frame coherency* – the assumption that consecutive game states are very similar. For example, a game character position (*p*) would be updated as $p = p + velocity*dt$, where *dt* is the time elapsed measured in the previous processing cycle. The simulation runs as fast as possible but the actual time step size depends on how fast the game loop runs – the faster the game loop runs, the smaller the time step size is. This affects directly the simulation sample rate. The simulation time step size is variable and depends on the current host hardware and current workload conditions.

In faster machines, the simulation time step will be smaller and the simulation sample rate will be higher. This measure makes it possible to bring *uniformity* to the simulation, effectively uncoupling the simulation and game loop sample rates. This also may lead to more accurate and smooth task results, which is a simple form of *adaptability*. In machines with more limited resources, the simulation and presentation will be less smooth but it will have a chance to keep real-time requirements. However, this approach may lead to oversampled tasks. In this case, these extra computations waste computational resources and power. When considering mobile game platforms, this issue may bring undesired power consumption. Although the simulation has uniformity, it remains as being non-repeatable and non-deterministic. Approaches based on elapsed time also lack *simulation control*.

Gregory et al. [12] reminds that when using the previous elapsed time in calculations for the next game state, these approaches implicitly assume that the time it will take to compute the next game state will be roughly the same as the previous one. If this is not the case, the game will also experience the "frame rate spike" [12]. Gregory et al. [12] mention that using a "running average" (the average elapsed time of a number of past frames) instead of just the previous elapsed time may lead to softening the frame rate spike effect.

The second approach to uncoupled time flow structures uses fixed sample rate loops, which creates an *explicit timeline* – the developer determines a target time step value and designs tasks to use this value. An explicit timeline enables *simulation control*, which makes it possible to pause, resume or run the simulation irrespective of real-time if desired. In this case, a game developer may run the game faster or slower than real-time for debugging purposes.

For example, a developer may define a target sample rate of 30 FPS and calculate the new position (*p*) of a game character as $p = p + velocity$, where *velocity* (the increment) is determined a priori based on the target sample rate. This approach turns the simulation *uniform*, *repeatable*, and *deterministic*. In fact, some tasks may work better when the sample rate is fixed, such as numerical integrators used in physics simulations [12].

In these approaches, the game loop runs all tasks and needs to wait before advancing the simulation to the next time step. For example, if the simulation sample rate is 30 FPS (i.e. time step is 33ms) and all task processing takes 10ms to complete, the game loop needs to wait for 23ms before advancing the simulation to the next time step. The extra available time make it possible to implement simulation *adaptability*, by using available extra time to improve simulation complexity and presentation quality. In these examples, all tasks have the same sample rate. In practice, the developer may determine different fixed sample rates for each type of task.

The simulation sample rate should be chosen carefully in order to avoid aliasing artifacts in the simulation. For example, a physics simulation with inadequate sample rates (e.g. large time step size) may appear to the player as being "jittery", "jumpy" or presenting unnatural behavior.

2.4 Interactivity

Interactivity is inherent to digital games, but it is a difficult term to define due to its subjectivity. Kiousis [13] conducted a thorough survey on several approaches to interactivity, leading to an operative definition of this concept based on three groups of concepts: *technological structure of the media used* (speed, range, timing flexibility, and sensory complexity), *communication settings characteristics* (third-order dependency and social presence), and *individual's perception* (proximity, perceived speed, sensory activation, and telepresence).

Kiousis' definition [13] provides hints about important concepts to consider in addressing interactive characteristic of a game loop. As the underlying structure of game simulations, we understand that the game loop is related to the *speed* and *perceived speed* concepts. The reason is that the game loop may affect directly these properties because it is the underlying structure of a game simulation. We understand that the other properties are dependent on the particular application that is built upon a game loop. The *speed* concept (technological structure group) refers to the (objective) rate which information enters the interactive system. For example, a mouse-based input system might be able to detect at most 15 clicks per second. The *perceived speed* concept (individual's perception group) refers to how a user perceives response times from the mediated environment.

3 GAME LOOP PROPERTIES

This section presents important properties related to *simulation*, *real-time*, and *interactivity* (Section 2). An adequate game loop model should address these properties, summarized in Table 1.

3.1 Simulation

This section presents properties that are directly related to game simulations.

3.1.1 Well-defined simulation model

A well-defined simulation model means that: 1) the simulation defines *target sample rates* for all tasks.; and 2) the simulation defines an *explicit timeline*, making the simulation *repeatable* and *deterministic*.

Game loops that do not address this property either do not define simulation time or define it implicitly. (e.g. coupled time

flow structures and uncoupled time flow structures based on elapsed time). In both cases, the simulation runs as fast as possible and becomes non-deterministic.

Simulation	Real-time	Interactivity
Well-defined simulation model	Uniformity	Responsiveness
Consistency	Resilience	Registration speed
Adaptability	Adequate use of physical resources	
	Energy management	

Table 1. Game loop properties

3.1.2 Consistency

Consistency in simulations is a key factor to keep players immersed in a game environment [14], which contributes to maintain the game illusion. We consider the game simulation as consistent if:

1. All game tasks are processed considering the ordering defined by task dependencies; and
2. The simulation does not produce artifacts that players may perceive as incorrect or inconsistent with the virtual world. This applies especially for physics simulations.

Coupled task flow organizations are able to produce consistent simulations as the task flow ordering is established by their structure (a serial pipeline). This is also the case of uncoupled task flow organizations that are single-threaded. However, in parallel game loops (Section 4) some tasks may finish before than others, and thus these task organizations require synchronization policies to ensure that task dependencies are addressed.

Another example involving consistency relates to how some tasks operate. For example, physics simulations in game environments may produce errors or inconsistent results if they are not modeled adequately. For example, players may perceive game objects as penetrating each other during game play. This might happen due to several reasons, such as inadequate time structure (e.g. inadequate step size, variable time step sizes, and aliasing artifacts) and problems in the collision detection algorithm.

3.1.3 Adaptability

This property refers to the capacity of a game loop to change simulation complexity and presentation quality according to the host hardware. For example, a game may be designed to provide a default game experience quality. While running, the game may discover that the host hardware exceeds the minimum system requirements to run the game. In this case, the game may improve simulation complexity and presentation quality by taking advantage of the extra computational resources, which may improve the game experience.

Adaptability also means the capacity of a game loop to change simulation complexity and presentation quality *dynamically* according to task load, in order to keep the game running in real-time. For example, if the system experiences momentarily high task load the game might reduce simulation complexity to keep the simulation running in real-time. This is different from adjusting presentation settings before the runs through configuration interfaces.

3.2 Real-time

This section presents properties that directly affect the real-time property of a game simulation.

3.2.1 Uniformity

This property refers to the capacity of a game to keep the simulation running in real-time regardless of different hardware configurations. For example, when two machines of different hardware configurations run the game side by side, the simulation appears to be in the same pace (real-time) in both of them. When this property is not addressed, the game may run faster than wall-clock time in more powerful machines and slower than wall-clock time in machines with limited capacity. In both situations, the game does not run in real-time. Game loop models that do not address uniformity do not have a well-defined simulation model (Section 3.1.1).

The coupling between the simulation and the frame rate results in coupled game loop models lacking simulation uniformity, which means that the simulation probably will not run in real-time, unless the simulation is carefully designed for a specific hardware. This was the case of games designed for video-game consoles, which had fixed hardware configurations.

3.2.2 Resilience

This property refers to the capacity of a game to maintain the simulation running in real-time when there are task delays, which may result in performance degradation. Resilience requires detecting problematic issues (e.g. performance degradation) concerning real-time pace and applying *adaptability* (Section 3.1.3) approaches. Players may perceive performance degradation as glitches in presentation, unresponsive input, animations lacking smoothness, and general slowness, among other effects. A game loop can be considered resilient if it addresses these issues before the player is able to perceive them or if it is able to recover quickly from performance degradation.

3.2.3 Adequate use of computational resources

This property relates to the fair use of computational resources. For example, an AI character may require 10 time steps per second to produce adequate results. Processing it at a higher sample rate will not improve its behavior. In this situation, the extra processing wastes computing resources. Another example are animations calculated and presented more than 60 times per second, as players probably will not perceive noticeable improvements. The game should adequate task processing up to a threshold that represents meaningful (and perceivable) results for players. Otherwise, the extra computations waste computational resources and energy.

3.2.4 Energy management

A consequence of inadequate use of computational resources is poor energy management. Research on energy management in games is under-explored – some works that explore this issue are [15]–[20]. Energy management considerations are even more important for games that run on mobile devices (e.g. smartphones), which operate on batteries that have limited autonomy.

3.3 Interactivity

When the simulation has low sample rate, players may experience severe input delay, which affect r*esponsiveness* and *registration speed*. In practice, it is hard to foresee if a game loop is able to address these properties because interactivity depends on factors that are application-dependent, such as the size of the virtual environment and current task work load. However, the game loop

may apply measures to minimize issues that might impact interactivity, such as employing *adaptability*, *resilience*, and defining an *adequate simulation model*.

3.3.1 Responsiveness

A game loop is responsive if it provides feedback or responses to players fast enough so that players do not perceive lags or delays regarding input. The notion of responsiveness should be considered from the player point of view, not being an objective measure.

3.3.2 Registration speed

The registration speed property is equivalent to the notion of "speed" that Section 2.4 discusses. Steuer [21] defines this concept as "*the rate at which input can be assimilated into the mediated environment*", claiming that this property is an essential factor that contributes to interactivity in virtual environments.

Chen and Thropp [22] cite several factors that contribute to lags in virtual environments when considering input, Some of these factors can be correlated to the registration speed property, such as: 1) input device sampling rate, 2) the time it takes to transfer the input information from a device to the computer, and 3) the time required to process raw device data before delivering the data to the application – for example, filtering sensor data (e.g. accelerometers, Kinect).

3.4 Game loop properties and time flow structures

As coupled task flow organizations define the time flow structure implicitly, they do not have a well-defined simulation model. Coupled task flow organizations are able to produce consistent simulations as the ordering of game task flow is established by their structure, which takes form as a task pipeline. In these organizations, the simulation simply runs as fast as possible, so they do not address adaptability.

The coupling between the simulation and the frame rate results in lacking simulation uniformity, which means that the simulation probably will not run in real-time, unless the simulation is carefully designed for a specific hardware. This was the case of games designed for video-game consoles, which had fixed hardware configurations. Running the simulation as fast as possible makes these models waste computational resources and power, failing to address these properties: Adequate use of computational resources and energy management. Coupled organizations are not resilient and are highly susceptible to frame rate variations. Valente et al. [1] provide several examples of basic coupled models.

Considering uncoupled task flow organizations (single-threaded or multi-threaded), there are many variations that differ mainly by implementation and parallelization techniques. Regarding game loop properties related to real-time, uncoupled task flow organizations generally address the uniformity property. However, not all uncoupled task flow organizations address these properties: adequate use of computational resources, energy management, and resilience. Considering game loop properties related to simulation, some uncoupled task flow organizations have a well-defined simulation model. For example, there are task flow organizations that use a time flow structure based on a fixed sample rate loop and there are task flow organizations that use an event-based timeline. On the other hand, there are uncoupled task flow organizations that address this property partially, such as the ones that use time flow structures based on elapsed time.

Valente et al. [1] presents several uncoupled models based on single-threaded approaches ("single-threaded uncoupled model", "fixed-frequency uncoupled model") and multi-threaded approaches ("multi-threaded uncoupled model", "asynchronous functional parallel model", "synchronous function parallel model"). This latter group represents simple approaches to introduce concurrency in game loop models. Figure 3 illustrates the "fixed-frequency uncoupled model" that contains a simulation state that runs at fixed-frequency.

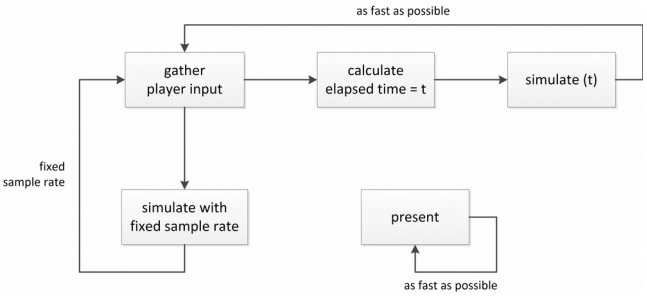

Figure 3. Fixed-frequency uncoupled model [1]

4 Task flow organization: A concurrency view

A task flow organization (i.e. game loop model) arranges game tasks as a graph or closed loop, which the simulation traverses continually to run all tasks. According to the time flow structure (Section 2.3), these task flow organizations may coupled (essentially single-threaded) or uncoupled (single-threaded or multi-threaded). Task flow organizations based on single-threaded approaches have been the norm in game development for many years. This situation has changed as currently multi-core CPUs are widespread in a variety of game-capable devices, such as PCs, dedicated video-game consoles, and mobile devices. Also using GPUs for non-rendering tasks has become commonplace.

Hence, this section explores task flow organizations based on heterogeneous and homogeneous processing, using a concurrency view The homogeneous processing group corresponds to models that implement concurrency using multi-threaded approaches in systems where all processing hardware are identical (i.e., they have the same architecture). An example is a system where all game processing occurs in a single multi-core CPU. The heterogeneous processing group corresponds to task flow organizations where the game runs on heterogeneous hardware. For example, in these approaches some tasks may run in a multi-core CPU, while other tasks may run on a GPU (as GPGPU).

4.1 Homogeneous processing view

The task organizations in this section describe concurrency in multi-core CPUs as cyclic task-dependency graphs. Each node in the graph represents a computational task and the graph edges represent task dependencies (flow dependencies or data dependencies). In these approaches, there are dedicated worker threads that run tasks in their own loop. These organizations have an entity known as the "task manager" that is responsible for distributing tasks to these worker threads. The worker threads may run any kind of game task and may be assigned to any CPU core. It seems that all task organizations approaches that we present in this section use uncoupled time flow structures, but this is not clear because the main focus in these research works is discussing how to perform concurrent task management. However, using a task scheduler may help in addressing adaptability, as the scheduler may be applied to improve scalability and task adaptation, as well as load balancing. Also, these approaches emphasize the quest for trying to achieve the best possible performance and do not provide further details on the game simulation model.

El Rhalibi et al. [6] presented an earlier framework to model games as cyclic task-dependency graphs, using a scheduler to run

game tasks in multiprocessor architectures. The model by El Rhalibi et al. [6] uses three concurrent threads, where each thread organizes simulation and rendering tasks according to task dependencies. El Rhalibi et al. [6] argue that their model scales well because it is able to allocate as many processing cores as they are available. Performance is limited by the amount of data processing that can run in parallel. An important issue is how to synchronize communication of objects running in different threads. El Rhalibi et al. state that the biggest drawback of this model is the need to have components designed with data parallelism in mind.

Tulip et al. [23] presents a related approach based on an acyclic tree service by a thread pool. The tasks that need to run in the game are broken down in the tree, where each tree node represents a single task. Each node in the tree has a number that represents its scheduling order. Some nodes may have the same number, which means that these nodes may be processed in parallel. During runtime, the game loop traverses the tree to extract tasks that are ready to run, placing them on a queue serviced by a thread pool. A task is not allowed to run until all tasks that have lower ordering (in the same parent node) are finished. It seems that the task tree is built statically – before the game loop runs.

Best et al. [24] propose Cascade, which is a framework for parallel game programming. The central concepts in Cascade are tasks, instances, task graph, and the task manager. The task computations are encapsulated in objects. The developer is responsible for instantiating these objects and specifying tasks dependencies using the Cascade API, which happens before the game application is run. This measure creates the task graph, statically.

AlBahnassi et al. [5] elaborate the idea of task graphs by proposing a solution that creates the task graph dynamically, as the Sayl design pattern. The Sayl design pattern is based on three main concepts (tasks, task dependencies, and scheduling) and has two main parts: a front-end and a back-end. Task and task dependencies comprise the front end, while task scheduling is part of the back end. The front end follows a pattern where the developer models task computations through method calling with parameters. The developer uses the parameters to model task (data) dependencies. The parameters can be evaluated in parallel (by several threads), and when all parameters are ready, the task is considered as ready for scheduling because the data dependencies have been resolved. The back-end has a task scheduler that maintains a task queue and maps tasks to the various CPU cores for execution. In Sayil, the task graph is built dynamically through the evaluation of parameters and dispatching tasks.

AlBahnassi et al. [5] state that their dynamic task graph approach is able to run tasks belonging to several frames in parallel. In other words, the approach defined by Sayil makes it possible to start computing the next game state before the current game state is finished, which is not possible in approaches based in static graphs such as the ones by [24], [23], and [6]. The task dependencies are enforced by the Sayil design.

When comparing these four approaches ([5], [6], [23], [24]), the dynamic task graph approach seems to bring opportunities to improve *resilience* and input *responsiveness* in games. For example, the game might try starting computing the next game state in advance (if possible) to address these properties. Accomplishing this will depend on the current workload and current task dependency complexity. The dynamic approach is more memory friendly than the static approach as the memory required for the task graph is allocated dynamically, which is not the case in static task graph approaches. This observation may lead to better addressing adequate use of computational resources.

4.2 Heterogeneous processing view

In this view, the task organization distributes some tasks to CPUs and some tasks to a GPU (as GPGPU). This section presents basic task flow organizations using GPGPU in Section 4.2.1. In these basic task flow organizations, the tasks that run on the GPU are pre-determined ("hard-coded"). Section 4.2.2 presents task flow organizations that implement some kind of task distribution among processors of different architectures. In these task organization flows, the same task may run either on a CPU or on a GPU. Section 4.2.3 presents a game loop architecture that implements *resilience* and *adaptability*.

4.2.1 Basic task flow organizations using GPGPU

Zamith et al. [3] presented a game loop model that integrated GPGPU, named as "multi-thread uncoupled model with GPGPU" (Figure 4). The GPU is used as an auxiliary math co-processor to process part of physics simulations. The model has three threads: one for input acquisition and simulation, one for rendering (presentation), and one for GPGPU. The multi-thread uncoupled model with GPGPU uses explicit synchronization primitives to handle task dependencies.

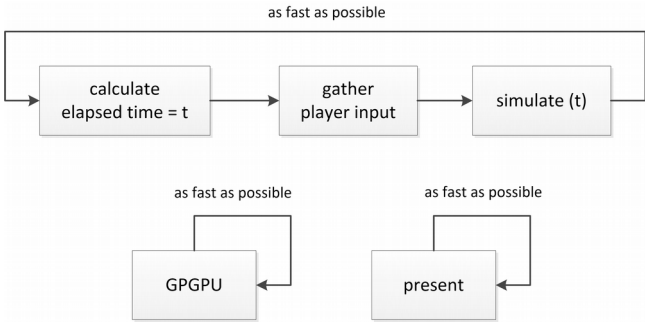

Figure 4. Multi-thread uncoupled model with GPGPU[3]

Joselli and Clua [25] proposed a task flow organization where the CPU and GPU switch roles in terms of main processing (Figure 5). This model uses an uncoupled time flow structure based on elapsed time, similar to the basic "multi-threaded uncoupled model" [1].

The GPU is responsible for processing the simulation and rendering stages. The simulation stage includes tasks such as AI, physics processing, and game logic processing. The CPU is used for gathering player input and running audio-related tasks.

In this model, there is a dedicated thread to process AI and a dedicated thread to process physics simulations. The implementation does not use explicit synchronization primitives due to the nature of GPGPU processing as a SIMD architecture. The GPU signals that a task is finishes only after all parallel execution streams are finished. In other words, the task flow does not proceed to the next task until all parallel streams of a tasks finishes.

These two models are based on the basic "multi-threaded uncoupled model" [1] and therefore present the same issues regarding game loop properties (Section 3.4). The model by Joselli and Clua [25] uses threads dedicated to some specific tasks. A straightforward improvement would be determining fixed sample rates for these tasks, which would define an explicit game timeline (Section 3.1.1).

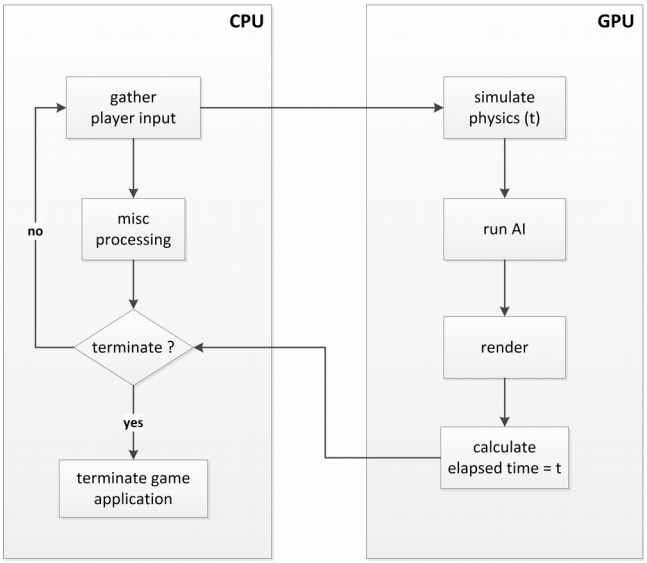

Figure 5. Task flow organization by Joselli and Clua [25]

4.2.2 Tasks flow organizations with task distribution

This section presents task flow organizations where the game loop is able to map task execution to processors of different architectures. This process may happen statically or dynamically. In both alternatives, the developer needs to provide different task implementations for each type of processor, since each hardware requires its own executable code

In terms of game simulation, the task distribution approach brings flexibility to run some game tasks. For example, if the game discovers at run time that it is possible to process some physics tasks on the GPU, it may take advantage of this resource to process these tasks faster, and use the extra time to process or to improve other game tasks. This may help in addressing game loop properties such as adaptability, resilience, and responsiveness, as well as lowering overall task processing time in order to meet real-time requirements.

An early effort to distribute task flows between CPU and GPU took form as the "adaptive game loop architecture" [26]. The "adaptive game loop architecture" uses an uncoupled time flow structure based on elapsed time and uses two parallel task flows that run on separate threads. The first task flow runs tasks related to input gathering, simulation, and rendering, which run only on the CPU. The second task flow contains simulation tasks that may run on CPU and the GPU. This architecture provides a task manager that is responsible for mapping tasks to both processors. The mapping policy is defined by a heuristic that the developer determines through a script file, which makes it possible to use different mapping strategies. Joselli et al. [26] provided a simple heuristics that compares the task running time on both processors and chooses the fastest one to run the task for the next time steps.

Joselli et al. [27] elaborated on the previous approach by creating a physics engine for games that provides alternative distribution schemes and runs the physics stage at a fixed sample rate (25ms) on its own thread. In this approach, the simulation stage comprises only the physics simulation task. Input and rendering tasks runs as fast as possible on a separate thread. This approach does not use explicit synchronization primitives to coordinate tasks. Instead, the implementation provides two buffers for the simulation to fill with data for rendering. The simulation uses one buffer at a time to output data for a given time step. While the simulation task is computing data, the rendering task presents data contained in the buffer that the simulation task is not using in that time step. The rendering task will keep on using the same buffer until the physics tasks is over, when they switch the buffers for the next frame. The approach by Joselli et al. [27] do not provide means to handle delays if the physics simulation takes too long, which impairs the resilience property.

AlBahnassi et al. [28] presented a more sophisticated approach to handle heterogeneous processors by using a scheduler to map tasks to different processors to minimize overall execution time. Their scheduler is based on these concepts: homogeneous processor groups, work stealing algorithm, and an arbiter. The homogeneous processor groups gather processors that have similar characteristics (as the cores of a CPU). The work stealing is the policy they have chosen to distribute tasks among processors of the same homogeneous group. Finally, the arbiter decides which homogeneous group is best suitable to process a given task, considering workload, execution times, data locality, and data transfer rates. The tasks are organized as dynamic task graphs (Section 4.1).

Similar to other approaches for heterogeneous processors, the developer needs to specify which processors are able to run the task and needs to provide alternative implementation for all the target processors. AlBahnassi et al. [28] provide a set of experiments that use different hardware combinations, including single-threaded, CPU-only, and CPU+GPGPU. Although the experimental results are interesting, it is not possible to determine the simulation model used in these tests.

4.2.3 Task flow organizations based on resilience

Zamith et al. [29] presented a game loop architecture that adjusts task execution according to hardware characteristics (CPU and GPU), game task characteristics, and current workload.

This model implements resilience by applying a tardiness policy. Zamith et al. [29] define this policy as *"a technique used as a metric to calculate task delays or earliness given a target time step, helping applications to satisfy real-time requirements"*. This metric is calculated as *t_e/t_p*, where *t_e* is the measured total task processing time and *t_p* is the predefined target processing time of all tasks (i.e. a *deadline* for the entire simulation step). This tardiness metric ranges from 0 to ∞. If this calculated metric is less than 1, it means that all tasks finished before the deadline and there is extra processing time available. In this situation the game may improve task quality to harness this extra time. On the other hand, if this metric is greater than 1, it means that task processing time was longer than the deadline, and therefore tasks need to reduce their processing requirements. In both situations, the architecture applies an *adaptability* approach.

To implement the *adaptability*, the architecture classifies game tasks into two types: divisible and indivisible tasks. The former represents that can be broken down in sub-tasks to be executed in consecutive time steps. The latter represents tasks that must be computed entirely on a single time step (due to hardware restrictions, they cannot be broken down in smaller parts).

5 CONCLUSIONS

The game loop is a term we use to describe the dynamics related to task execution and task flow organization in games. We conducted an investigation to understand the inner workings of game simulations through a literature review on game loops, game simulations, and our own experience on game development. In this investigation, we identified properties related the main characteristics of game loops, considering that game simulations must be *discrete*, *real-time*, and *interactive*. As discrete simulations, a game loop is concerned with well-defined *simulation model*, *consistency*, and *adaptability*. As real-time simulations, a game loop is concerned with *uniformity*, *resilience*,

adequate use of computational resources, and *energy management*. As interactive simulations, a game loop is concerned with *responsiveness* and *registration speed*.

Game development traditionally has been characterized by the quest for ultimate computational performance. This quest started a long time ago because earlier games ran on hardware with very limited capabilities (when compared to current hardware) where rendering was a very costly operation, placing a huge bottleneck in game development. This issue has led to the first (and simplest) game loops that use implicit time flow structures. This quest still exists nowadays, but blindly trying to fulfill it does not help in improving games. For example, when achieving the best possible performance, a game might apply *adaptability* policies to improve simulation quality when there is extra time available (e.g. increasing complexity of AI behavior physics modeling). Otherwise, if game processing is too heavy, a game may apply *adaptability* to reduce simulation complexity, which would benefit *responsiveness* and *registration speed*. These measures may help in implementing *resilience* policies, which would help the game to tolerate task delays. When a game simply runs as fast as possible, as a result of aiming at the best possible performance, the game may *waste computational resources* and increase *energy consumption*. Also if the game does not have a *well-defined simulation model*, it probably will not run in real-time. We also notice that there are few publications that address energy management in games. This research field remains open and under-explored.

The first multi-threaded game loops applied straightforward parallelism concepts such as functional and data parallelism by dedicating threads to process specific types of tasks in the first case or entire sets of related data in the second case. However, during our investigation we noticed a trend consisting of new task flow organizations that consider dynamic task graphs (built according to task dependencies) and task schedulers. These task schedulers take advantage of multi-core CPUs in order to have tasks running in parallel. A related trend is running games in heterogeneous environments consisting of multi-core CPUs and GPUs for general-purpose processing (GPGPU).

With parallel hardware (in form of multi-core CPUS and GPGPU) becoming increasingly common, we believe that in the near future a greater interaction among the areas of parallel computing and game development will emerge. Considering the new scenario, future works on game loops may explore topics such as:

- New game loop models based on parallelism;
- Task load balancing techniques;
- Simulations models with fixed sample rate that consider multiple processors;
- Simulation models focusing on heterogeneous environments;
- Task flow implementations that implement policies to tolerate task delays;
- Methods to allow processing more than one frame simultaneously, when it is possible. For example, a task organization might start processing in advance some tasks belong to the next frame if there is free time available, which could help in tolerating task delays and implementing resilience policies;
- Adaptability policies.

ACKNOWLEDGMENTS

We would like to thank CAPES, CNPq, FINEP, and NVIDIA for the financial support to this research paper.

REFERENCES

[1] 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.

[2] I. García and R. Mollá, “Videogames decoupled discrete event simulation,” *Computers & Graphics*, vol. 29, no. 2, pp. 195–202, Apr. 2005.

[3] M. Zamith, E. Clua, A. Conci, A. Montenegro, P. Pagliosa, and L. Valente, “Parallel processing between GPU and CPU: Concepts in a game architecture,” in *Computer Graphics, Imaging and Visualisation, 2007. CGIV ’07*, 2007, pp. 115–120.

[4] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, 1 edition. New York: Wiley-Interscience, 2000.

[5] W. AlBahnassi, S. P. Mudur, and D. Goswami, “A Design Pattern for Parallel Programming of Games,” in *14th International Conference on High Performance Computing and Communication*, 2012, pp. 1007–1014.

[6] A. El Rhalibi, D. England, and S. Costa, “Game Engineering for a Multiprocessor Architecture,” in *Changing Views: Worlds in Play*, 2005.

[7] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, New York, NY, USA, 1967, pp. 483–485.

[8] J. L. Gustafson, “Reevaluating Amdahl’s Law,” *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[9] S. Cook, *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs*, 1 edition. Amsterdam ; Boston: Morgan Kaufmann, 2012.

[10] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*, 2 edition. Amsterdam; Boston; Waltham, Mass.: Morgan Kaufmann, 2012.

[11] R. Parent, *Computer Animation, Third Edition: Algorithms and Techniques*, 3rd ed. Amsterdam: Morgan Kaufmann, 2012.

[12] J. Gregory, J. Lander, and M. Whiting, *Game Engine Architecture*. Wellesley, Mass: A K Peters/CRC Press, 2009.

[13] S. Kiousis, “Interactivity: a concept explication,” *New Media Society*, vol. 4, no. 3, pp. 355–383, Sep. 2002.

[14] C. Hecker, “Physics in Computer Games,” *Commun. ACM*, vol. 43, no. 7, pp. 34–39, Jul. 2000.

[15] Y. Gu and S. Chakraborty, “Power Management of Interactive 3D Games Using Frame Structures,” in *21st International Conference on VLSI Design, 2008. VLSID 2008*, 2008, pp. 679–684.

[16] S. Chakraborty and Y. Wang, “Multimedia Power Management on a Platter: From Audio to Video & Games,” in *Proceedings of the 16th ACM International Conference on Multimedia*, New York, NY, USA, 2008, pp. 1165–1166.

[17] B. Anand, K. Thirugnanam, L. T. Long, D.-D. Pham, A. L. Ananda, R. K. Balan, and M. C. Chan, “ARIVU: Power-aware Middleware for Multiplayer Mobile Games,” in *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, Piscataway, NJ, USA, 2010, pp. 3:1–3:6.

[18] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan, “Adaptive Display Power Management for Mobile Games,” in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, 2011, pp. 57–70.

[19] A. Bhojan, Z. Qiang, and A. L. Akkihebbal, “Energy Efficient Multi-player Smartphone Gaming Using 3D Spatial Subdivisioning and Pvs Techniques,” in *Proceedings of the 3rd ACM International Workshop on Interactive Multimedia on Mobile & Portable Devices*, New York, NY, USA, 2013, pp. 37–42.

[20] M. Zamith, L. Valente, M. Joselli, J. R. Silva Junior, E. Clua, and B. Feijó, “A Game Architecture Based on Multiple GPUs With Energy Management,” in *Proceedings of SBGames 2013*, São Paulo, 2013, pp. 54–63.

[21] J. Steuer, “Defining Virtual Reality: Dimensions Determining Telepresence,” *Journal of Communication*, vol. 42, no. 4, pp. 73–93, Dec. 1992.

[22] J. Y. C. Chen and J. E. Thropp, “Review of Low Frame Rate Effects on Human Performance,” *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 37, no. 6, pp. 1063–1076, Nov. 2007.

[23] J. Tulip, J. Bekkema, and K. Nesbitt, “Multi-threaded Game Engine Design,” in *Proceedings of the 3rd Australasian Conference on Interactive Entertainment*, Murdoch University, Australia, Australia, 2006, pp. 9–14.

[24] 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*, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin Heidelberg, 2009, pp. 912–923.

[25] M. Joselli and E. Clua, “GpuWars: Design and Implementation of a GPGPU Game,” in *2009 VIII Brazilian Symposium on Games and Digital Entertainment (SBGAMES)*, 2009, pp. 132–140.

[26] M. Joselli, M. Zamith, E. Clua, A. Montenegro, R. Leal-Toledo, A. Conci, P. Pagliosa, L. Valente, and B. Feijó, “An adaptative game loop architecture with automatic distribution of tasks between CPU and GPU,” in *Proceedings of SBGames’08: Computing Track - Full Papers*, Belo Horizonte, 2008, pp. 115–120.

[27] M. Joselli, E. Clua, A. Montenegro, A. Conci, and P. Pagliosa, “A New Physics Engine with Automatic Process Distribution Between CPU-GPU,” in *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*, New York, NY, USA, 2008, pp. 149–156.

[28] W. AlBahnassi, D. Goswami, and S. P. Mudur, “Arbiter Work Stealing for Parallelizing Games on Heterogeneous Computing Environments,” in *Proceedings of the High Performance Computing Symposium*, San Diego, CA, USA, 2013, pp. 16:1–16:9.

[29] M. Zamith, L. Valente, B. Feijó, M. Joselli, and E. Clua, “Exploring parallel game architectures with tardiness policy,” in *Proceedings of SBGames 2015*, Teresina, 2015, pp. 94–103.