Real Time Game Loop Models for Single-Player Computer Games

Luis Valente[1]
Aura Conci[1]
Bruno Feijó[2]

[1]Universidade Federal Fluminense – Instituto de Computação
{lvalente,aconci}@ic.uff.br

[2]ICAD/IGames/VisionLab, Departamento de Informática – PUC-Rio
bruno@inf.puc-rio.br

Abstract

There are few academic works on game loop models and algorithms in the literature. Also, the literature lacks a comprehensive conceptual framework for real time game loops. This paper proposes a general classification for real time game loop models and presents an algorithm that improves one of the most common models for single-player computer games.

Keywords: *Real time loops, computer games*

1 Introduction

Real time systems have time restrictions to perform their work. In other words, the time required to execute all tasks must not exceed some threshold; otherwise the system will fail [1].

Interactive real time systems have three main modules: input data acquisition, processing and presentation of results [2]. In computer games, the input system corresponds to input device management (such as keyboard, mouse, and joystick). The processing stage is responsible for taking decisions that affect the game state. The presentation stage presents the results from other stages, using audio and video.

During the game execution life cycle, if it is not possible to execute all those complex tasks below some time threshold, the interactivity of the game may not be acceptable. This issue characterizes computer games as a heavy real time application.

A common parameter, which an application may use to estimate performance, is the amount of frames per second (FPS) displayed on the screen. A frame represents an image that the application builds and displays on screen. A common accepted lower bound for FPS, in order to maintain interactivity, is 16 frames per second. A frame rate from 50 to 60 FPS is considered optimal.

A real time game loop model describes a particular way of organizing the execution of game tasks. This important issue is in the heart of real time applications. However, it is very difficult to find academic works on this subject. The works by Dalmau [2], Dickinson [6], and Watte [10] are amongst the few ones. Also, the literature lacks a comprehensive conceptual framework for real time game loops. This paper proposes a general classification for real time game loop models and presents an algorithm that improves one of the most common game loop models. Although some of the loop models can be used in any kind of game, the present paper focuses on single-player computer games. This research was performed during the specification of the Guff framework [3], which is a game development tool.

2 Real Time Loop Models

A real time game loop model is an approach to organize the execution of game tasks. The way these tasks are organized determines the game architecture and how the game runs in different machines.

While the game is running, the user perceives the stages of input data acquisition, processing, and rendering as occurring simultaneously. However, most computers today have only one processor and limited amount of memory, so it is necessary to arrange such tasks in order to simulate that parallelism and to maintain interactivity. In fact, this is the major motivation to elaborate real time game loop models.

The tasks controlled by a game loop can be divided in three stages:

- The input device query;
- The update stage;
- The presentation stage.

The first one is responsible for gathering user input. The second stage executes update tasks that determine the current game state, such as the animation interpolation, the physics simulation, the game AI (Artificial Intelligence), and the game logic (e.g. the application of the game rules). The third stage executes the scene rendering at a certain frame rate. In this section, we propose the definition of two main groups of loop models, based on the way the above-mentioned stages are interrelated:

- Coupled models;
- Uncoupled models.

2.1 Simple Coupled Model

The simplest approach to model a real time game loop consists in executing the update and presentation stages sequentially [2, 4]. Figure 1 depicts this model.

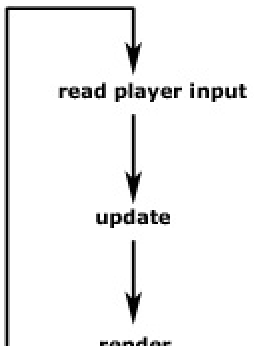

Figure 1: Simple Coupled Model

The main advantage of this approach is its simple implementation. Computer games that run in platforms with fixed configuration (such as videogame consoles) may apply this model with no further problems. For example, in order to calculate the position of game character in the game, let the displacement of this character be s, and let its current position be $p1$. A new position, $p2$, would be calculated as follows:

$$p2 = p1 + s$$

However, on platforms with variable configuration (such as PCs), this model will not work properly. The main problem of the Simple Coupled Model is that the game will not run in all machines uniformly. A computer with high processing power would run the loop more often. In this case, the game would run more quickly, but this does not mean it would provide a better experience for the user. In the former example of position updating, the effective game character displacement would depend on the machine where the game is running. This issue is of great importance in multiplayer games, where the players should share the same view from of the game (in other words, their views should be synchronized to each other).

Another problem related to the Simple Coupled Model is how the user perceives performance degradation. This issue may manifest itself if a game task lasts too long to run. The user may perceive this degradation easily, for example, as an animation that lacks smoothness.

2.2 Synchronized Coupled Model

A possible variation of the Simple Coupled Model is to synchronize the loop execution with some pre-determined frequency [4, 5]. Figure 2 illustrates this model called Synchronized Coupled Model.

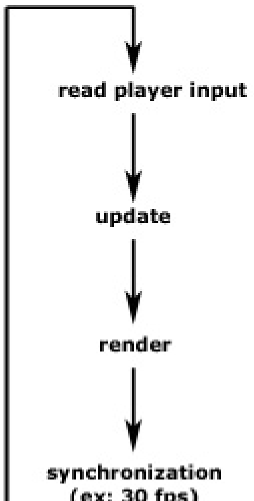

Figure 2: Synchronized Coupled Model

This variation is an alternative to bring uniformity to the game loop execution, among different machines. However, this approach poses an artificial limitation for the application. Even though different computers should run the game in a similar way under this model, a more powerful machine will not provide a better experience to the user than a lower-end one (such as more smooth animations). Therefore, this model wastes the processing power of higher-end machines. In other words, this model does not scale.

An alternative to solve this problem is to have the update and presentation stages independent of each other, what leads to the class of uncoupled models.

2.3 Multi-thread Uncoupled Model

A naïve idea for implementing a uncoupled model would be a simple separation of the update tasks from the presentation ones, in such a way that both tasks would be executed as fast as possible (Figure 3).

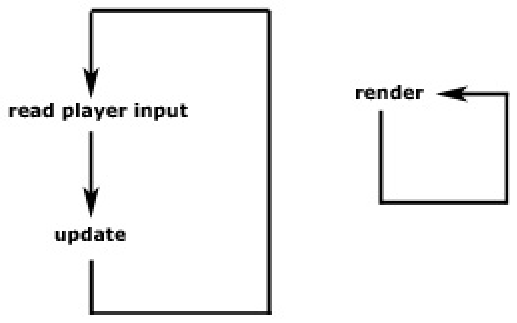

Figure 3: Separation of the main stages

However, this approach may cause different machines to run the game in diverse ways, as does the Simple Coupled Model. In order to avoid this behavior, it is necessary to uncouple the update stage from the frame rate.

This can be done by applying a time parameter to the update stage. This parameter corresponds to the time elapsed since the last update stage execution. This value would be applied in all calculations. For instance, if the current position of a game character is *p1* and its velocity is *v*, then the next value for its position, *p2*, would be:

$$p2 = p1 + v * elapsedTime$$

Thus, computers with diversified configurations will run the update stage correctly. A powerful machine would run the loop more often (using a lower delta value), which means it would provide a better experience to the user (more smooth animations, for example). A machine with low processing power would still provide the correct result, even if this means less quality. In this paper, the game loop model based on this approach is called Multi-thread Uncoupled Model (Figure 4).

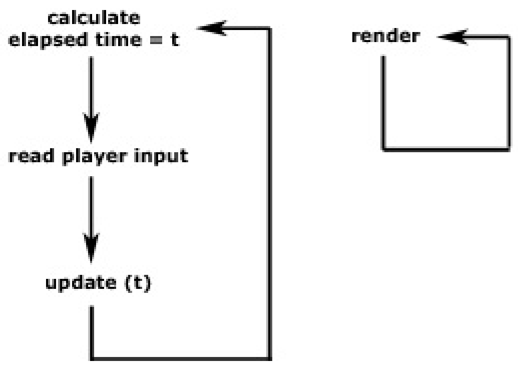

Figure 4: Multi-thread Uncoupled Model

The implementation of the Multi-thread Uncoupled Model involves typical issues of concurrent programming, such as explicit synchronization to data access. For example, if the update stage is responsible for animating a game character and explicit synchronization is not applied, then the presentation stage may access inconsistent data (if the update stage does not finish its processing on time).

2.4 Single-thread Uncoupled Model

An approach to avoid concurrency issues in uncoupled models is to simulate multi-threading as a single-thread process. Figure 5 illustrates this scheme, called Single-thread Uncoupled Model.

This implementation presents benefits from an uncoupled game loop model, without concurrency issues.

2.5 Fixed Frequency Uncoupled Model

Some remarks must be considered, regarding the uncoupled game loop models discussed so far. Up to this point, the update stage is responsible for all processing tasks that affect the current game state, such as game logic execution (*e.g.* game rules), Artificial

Intelligence, animation interpolation, and physics simulation (*e.g.* rigid body dynamics, collision detection, and collision handling). If the update stage executes all these tasks every time, it may waste computational power. This is because some tasks may not present significant results if they are executed sequentially, in a brief time interval. An example of this type of task is the game logic execution. On the other hand, there are tasks that may present better results if they are executed more often, like animation interpolation.

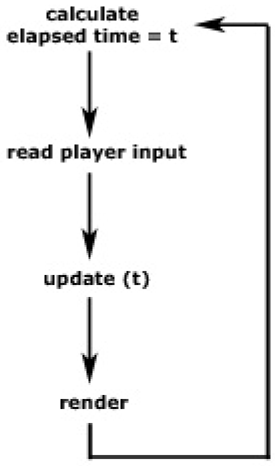

Figure 5: Single-thread Uncoupled Model

Thus, another approach to the question of game loops is to separate the update stage into two substages: one which may present better results if it runs more often, and another substage that should run less frequently. The second substage runs at a fixed frequency.

3 Deterministic Models

Figure 6 depicts a model that applies this approach, called Fixed Frequency Uncoupled Model in this paper. Under this model, the rendering loop would run as fast as possible.

An interesting consequence regarding the Fixed Frequency Uncoupled Model is that it makes the game execution deterministic. Generally speaking, update stages that depend on the system time are non-deterministic [6]. In this case, the time measurements performed in the update stage may report different results whenever the game runs. Many factors may contribute to this, such as the process load on the computer at the time the experiment is run. Therefore, if a dataset (as player input data, for instance) is supplied to the game, the end result which it presents (animations, images), may vary, rendering its execution non-deterministic.

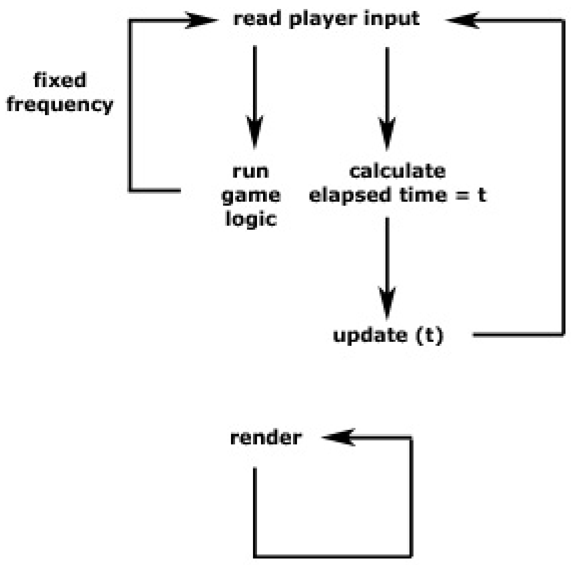

Figure 6: Fixed Frequency Uncoupled Model

There are some game functionalities that can take advantage from game execution determinism, such as replay feature, program debugging, and network module implementation [6].

The replay feature permits the game to play an event sequence (such as an animation), repeatedly. This feature is available on sports and racing games, for example. To create such sequence, it would suffice to record the input data from the player and supply them later to the game, in order to play the event sequence in real time. In the Fixed Frequency Uncoupled model, it is irrelevant if those data come from the player or are pre-recorded (as a file, for example).

The program debugging is a developer-only feature. Eventually, it is necessary to test the game and address errors related to the gameplay. If the game uses a deterministic game loop model, it is possible to reproduce errors more easily, by supplying the command sequence that leads to it.

In networked games, it is important that all users share the same view from of the game, in order to maintain consistency. A network module implementation may be easier if it uses a deterministic model, because the same command sequence always generates the same result. Hence, it is possible to synchronize the game views by sending the input data from the other players [6].

4 The Optimized Fixed Frequency Algorithm

In this paper, a new and efficient game loop algorithm is proposed, based on the Fixed Frequency Uncoupled Model (Figure 6). This algorithm was selected to be the heart of the Guff system. Guff is a game development system developed by the first author as a test bed for new algorithms and models [3]. As Guff is a general-purpose game development tool, it is necessary to implement a flexible real time game loop. The Guff system is described in Section 5.

The coupled model that Figure 1 illustrates would be a nice choice if Guff were designed for platforms with fixed configuration, such as videogame consoles. On such platforms, this model allows a game to run in a deterministic way. However, on platforms with varying configurations, this model is inadequate, since the game behaves differently on each platform.

The synchronized coupled model (figure 2) tries to improve the basic coupled model. However, it does not work well, because it makes all machines run the game in a similar way at the lowest level of quality.

The uncoupled models with a single update stage (figures 4 and 5) try to overcome the limitations that the coupled models impose, by supplying a time parameter to the update stage. The main idea is to apply the time parameter to all calculations, so that all machines run that stage correctly.

However, there are some drawbacks related to these versions of uncoupled models:

- These models are not deterministic, since they apply a time parameter (whose measurement possibly varies every time the game is run) in all calculations. This means the game is harder to debug and makes the implementation of some features more difficult;
- The time parameter complicates calculations performed in physics simulation (besides turning it into a non-deterministic simulation);
- The repeated execution of some tasks (such as Artificial Intelligence algorithms) in a very brief period of time may possibly waste processing power.

In order to address these issues, the Fixed Frequency Uncoupled Model (figure 6) splits the update stage into two parts: one which runs as fast as possible and another that runs at a pre-determined frequency.

This approach is a compromise between coupled and uncoupled models. It presents advantages from both the first group (determinism) and the second group (some tasks may present better results on higher-end machines). However, this is the most complicated model.

The improved Guff game execution model is based on the Fixed Frequency Uncoupled Model, as it is the most flexible among the available models. The Guff implementation features two aspects: pause awareness and early recovery from long delays.

The pause awareness feature means this game loop differentiates a pause request from a long delay. For example, a pause request applies in the following situations:

- The user decides to pause the game. Then it is necessary to stop running the simulation until the user decides to resume the game;
- It is necessary to switch to another task. For example, if the user invokes a menu while the game is running, it is necessary to pause the current simulation, until the user finishes using the menu.

Figure 7 depicts a situation where a pause request is issued. The simulation time should not be affected by the time the game remais paused, in order to maintain the game execution determinism. Therefore, it is necessary to restore the previous timeline when the game is resumed.

The early recovery from long delays feature means the game loop checks whether the fixed frequency update stage has spent more time than expected, just after it is executed, instead of waiting for the next loop iteration to verify this. The purpose of this approach is to keep the simulation time more accurate, if delays occur.

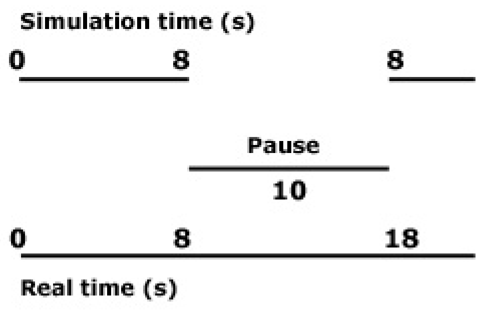

Figure 7: A pause request scenario

The following pseudocode presents the algorithm that implements the fixed frequency uncoupled model, in such a way that takes into account the features just discussed:

```
make lastFrameTime equal to the current
real time

while game is running {

 if there is a pause request
 {
   make pauseTime equal to the current
   simulation time
 }

 if there is a resume request
 {
   calculate total elapsed time due to
   game pauses: current real time -
   pauseTime

   queries for the current simulation
   time (like in the first if and in the
   next while loop) should return:
   current real time - pause elapsed time
 }

 if the game is not paused
 {
    while the current simulation time -
          lastFrameTime is greater than
          or equal to the updateStep
      {

         run the fixed frequency update
         stage

         update the lastFrameTime, by
         adding the updateStep

         check current simulation time to
         find out if the fixed frequency
         update spent more than
         updateStep to run
      }
 }

 calculate time elapsed since the last
 loop execution, at this point (time)

 execute the update stage for the
 calculated time

 execute the rendering stage
}
```

The game loops presented in the literature [2, 6, 10] do not distinguish between pauses and long delays. This means a game pause (which is interpreted as long delay) may create havoc in the game execution. This is the reason why the proposed algorithm is referred to as an optimized one.

In the former pseudocode, the update time step (`updateStep`) represents the desired time interval between two consecutive executions of the fixed frequency update loop. For example, if it is required to execute this loop at 20 Hz, this value would be 50 ms.

5 The Guff Framework (games-uff)

This section presents the Guff framework used in this work. Frameworks are a set of classes that form a reusable architecture for a family of systems [7].

The Guff framework was implemented with the C++ programming language and applies paradigms like object-oriented programming, design patterns [7] and automatic resource management [8]. This tool defines a reusable architecture for games and initiates a research field at Universidade Federal Fluminense (UFF) on game development. Guff comprises two main parts: the application layer and the toolkit.

The application layer defines the interfaces and program architecture, modeled as a state machine. The state machine implements a real time game loop model to run the application. The motivation for this approach is to represent each game level as a state, in order to ease their specification and maintenance. Figure 8 illustrates the Guff application layer class diagram.

The State class represents a single state, whereas the StateGroup class represents a group of states. The MasterState class represents the default parent state in Guff applications. The state classes have methods that correspond to events that the developer can handle. The class that is responsible for implementing the real time game loop model (StateAppRunner) delegates the event handling to the application, on behalf of the current state.

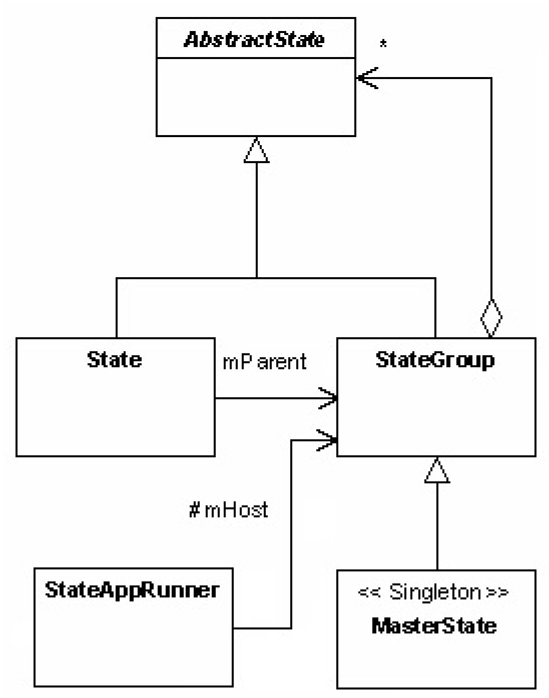

Figure 8: Guff application layer UML diagram

The toolkit stands for a set of auxiliary classes, which developers may use to build new applications. It provides functionalities related to visualization, automatic management of third party libraries, application configuration, input devices, Math, audio and utilities. The Guff visualization module has classes that handle the rendering of Quake 3 [9] scenes (often referred to as map files). Quake 3 is a first-person shooter [3] game. This type of game presents three-dimensional indoor scenes, often organized as mazes. The main motivation to use Quake 3 scenes is that it is relatively easy to find new maps and map editing tools related to this game.

6 Related Work

In [2], the author defines a multi-thread model called "twin-threaded approach" that can be identified as the Fixed Frequency Uncoupled Model. He defines the single-thread variation as "single-thread fully decoupled".

In [6], the author discusses a uncoupled model that separates the update and presentation stages. This model executes the update stage at a fixed frequency. The main objective of such model is to implement replay features in a game. The author also discusses benefits associated with networked games, when a game applies that model.

A model with a fixed frequency update stage similar to the one discussed in [6] is defined as "canonical game loop" in [10]. Traditional game development portals, such as [11] and [12], cite this reference.

The above-mentioned works [2, 6, 10] do not consider pause requests as a distinct event. The algorithm proposed in the present paper (Section 4) handles this situation, in order to provide a way to interrupt the simulation while maintaining its current timeline. This feature also helps to keep the game execution deterministic.

7 Conclusions

There are few academic works on game loop models and algorithms in the literature. Moreover, the well-known commercial games do not reveal details of their architecture. Also, the literature lacks a comprehensive conceptual framework for real time game loops. This paper proposes a general classification for real time game loop models and presents an algorithm that improves one of the most common game loop models. As far as the authors are concerned there is no such a comprehensive classification of game loop models in the literature.

The proposed algorithm provides a way to interrupt the execution of the fixed frequency update stage, and to restore the simulation timeline later. This feature makes it possible for the game to run as if there were no interruption, and preserves the determinism. The canonical algorithms [2, 6, 10] do not distinguish between pauses and long delays. This may disorder the game execution.

A future work would be a comparison of the proposed algorithm with top-level commercial games, what is not easy because of classified issues.

Acknowledgements. The authors thank the financial support of CAPES. The second author thanks CNPq (Grant PQ No. 302436/2004-9). The third author also thanks CNPq and FINEP for the research contracts CNPq Grant PQ No. 305982/2003-6, SEPIN-CNPQ-FINEP No. 01.02.0235.00 (Ref. 2425/02) and FINEP No. 01.04.0945.00 (Ref. 3110/04).

8 References

[1] Silberschatz, A., and P. Galvin, *Operating Systems Concepts*, 5th Edition, Addison-Wesley, Reading, 1998.

[2] Dalmau, D., *Core Techniques and Algorithms in Game Programming*, New Riders, Indianapolis, 2003.

[3] Valente, L., *Guff: Um Sistema para Desenvolvimento de Jogos*, Master's Thesis, Universidade Federal Fluminense, Niterói, 2005 [in Portuguese].

[4] Rollings, A., and D. Morris, *Game Architecture and Design: A New Edition*, New Riders, Indianapolis, 2003.

[5] LaMothe, A., *Tricks of the Windows Game Programming Gurus*, Sams, Indianapolis, 1999.

[6] Dickinson, P., *Instant Replay: Building a Game Engine with Reproducible Behavior*, Available at http://www.gamasutra.com/features/20010713/dickinson_01.htm (03/20/05).

[7] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, 1995.

[8] L. Valente, and A. Conci, "Automatic Resource Management as a C++ Design Pattern", *Simpósio Brasileiro de Jogos para Computador e Entretenimento Digital,* Curitiba, 2004, CD-ROM.

[9] id Software, *Quake III Arena*. Available at http://www.idsoftware.com/games/quake/quake3-arena (04/25/2005)

[10] Watte, J., *Canonical Game Loop*, Available at http://www.mindcontrol.org/~hplus/graphics/game_loop.html (03/20/05).

[11] *Flipcode.org*, Available at http://www.flipcode.org (03/20/05).

[12] *Gamedev.net*, Available at http://www.gamedev.net (03/20/05).