Poster Proceedings of Pacific Graphics (2008)
T. Igarashi, N. Max, and F. Sillion (Editors)

Effective Load Management Technique for AI Characters in Games

Brian Tan †1 and Gabriyel Wong ‡1

1Nanyang Technological University, Singapore

Abstract

Creating large populations of AI characters for game environment is a major challenge because either insufficient CPU processing time is available or it is difficult to balance the computational needs of the game AI against the requirements of other game components. We present a novel technique that allows games to manage the functional updates of AI characters efficiently during runtime. The technique is an enhancement of the elastic task model and scheduling [BLCA02] which allows the total CPU utilization of the AI characters to be adjusted to the current workload of the game. We exploited the elastic nature of the technique to provide Level of Detail (LOD) effect for the AI characters. A prototype implementation on Microsoft Xbox 360 hardware is described in this paper.

Categories and Subject Descriptors (according to ACM CCS): F.2.2 [Nonnumerical Algorithms and Problems]: Sequencing and Scheduling I.2.1 [Artificial Intelligence]: Games

1. Introduction

Artificial Intelligence (AI) characters are computer-controlled virtual characters in games that are part of many engaging games. Depending on the design of the game, these AI characters could range from herds of wildlife in natural environments to pedestrians or vehicular traffic in urban settings.

Many real-time virtual worlds do not look natural either because there are too few AI characters or the behaviours of these characters are too rudimentary until they seem unrealistic to the human players [NG05]. This is because human players are accustomed to vibrant, dynamic and busy real world environments resulting in similar virtual environments which are sparsely-populated to look unnatural.

In this paper we present a novel technique to manage the computational load of a large population of AI characters which is independent of the actual AI routine utilized and provide the benefit of adaptive response to transient fluctuations in the CPU utilization of the game.

† TANK0108@ntu.edu.sg
‡ ckwong@ntu.edu.sg

2. Related Works

It is difficult to implement a large number and variety of AI characters in games when the computational resources available for game AI processing is limited. Level of detail (LOD) approaches have been used in games as solutions for handling this particular problem. Similar to applications of LOD in computer graphics, LOD for game AI is a view-dependent technique that relies on hiding computationally expensive details that the player cannot detect by using impostors or proxies for the less-visible characters [DHOO05, KDC*08], simplified shortcuts for the gameplay processing required [Bro02] or switching between reactive and proactive behaviours [NG05]. These technique share one common feature, they create different LOD by providing different versions of the same targeted feature (geometrical model, behavioural simulation or animation) with different degrees of simplification.

Implementing LOD in the manner described above requires the game developer to create different versions of the AI routines in addition to the originals. The amount of development effort is thus increased multiple-fold which makes it challenging for game developers facing tight deadlines. An alternative approach is to design the AI workload as processing tasks that can be managed and scheduled efficiently during runtime [WM00, Ale02, McL02]. This has the added advantage of good control over the amount of processing time consumed by the game AI.

Much research work have been carried out in the field of real time systems and task scheduling. We see the opportunity to tap this reserve of research work for the purpose of creating a load management system for AI characters. We are particularly interested in the elastic task model introduced and studied extensively in real-time system literatures [BA02, BLCA02, CHL06], which has the flexibility of adapting to changes in the overall workload.

3. Requirements for scheduling AI execution in games

This section outlines the different challenges facing game developers when they need to optimize the performance of AI characters in games. The technique proposed in this paper is designed to meet the requirements outlined in this section.

3.1. Conformance to timing constraints

Frame rate of games need to be maintained at a minimum of 30 display frames per second if not higher, as lower frame rates decreases the visual quality and responsiveness of the game. In order to meet this requirement, the time allowed for all manners of processing required has to be limited to 33 milliseconds or less. The processing time allowed per frame is divided and distributed to each of the different game tasks; physics simulation, gameplay logic, player inputs, rendering and game AI. Each of the game tasks or components must conform to its given CPU time budget for the game to maintain a stable and high frame rate.

Therefore, it is important that any solution aimed at tackling the issue of game AI performance must be able to make the total processing time of the AI characters conform to the timing constraint imposed upon it. Failure to do so would have a negative effect on the frame rate of the game.

3.2. Effective utilization of the limited CPU processing time allocated

Typically, only 30 percent of the CPU time budget is allocated to game AI as cited in the 2000 Game Development Conference (GDC 2000) AI polls [Woo00]. This limited allocation is not enough to create the beautiful and vibrant virtual environments demanded by players. Large numbers of AI characters of different types and believable behaviours are needed to effectively populate a virtual environment thus making it an uphill struggle to provide the best possible *quality of service* while conforming to the limited CPU time budget. The *quality of service* can be any performance metric that the developer wish to optimize: animation quality, interactions with the AI characters, minimum number of errors et cetera. The key lies in distributing the limited CPU time budget to maintain the visual quality and interactiveness of the AI characters. This calls for a level of detail approach or a technique that would provide a similar effect.

3.3. Compatibility with different AI techniques

The different variety of AI characters also calls for different techniques to be used for its implementation. For example, ReynoldŠs flocking techniques [Rey87, Rey06] may be suitable for flocks of birds, herds of herbivores or schools of fishes, but human crowds and groups would utilize techniques from a totally different pool of research work [MUAT05, TCP06, RD05]. The same can be said for ambient traffic and pedestrians [BCKW98, dSM06]. Taking into consideration the wealth of AI techniques that can be used by the game developers, any solution that intends to be applicable to the different varieties of AI characters have to be independent of the underlying techniques. It should make little or no assumptions about the nature of the underlying AI code.

3.4. Runtime response to transient fluctuations in game workload

Even as game developers make an effort to adhere to the specified CPU time budgets, fluctuations in processing time can still occur during runtime. These fluctuations could originate from within the game AI itself or from other game components. Computationally heavy game components like physics simulation, rendering and game AI are vulnerable to major changes in the game scene. For example, a major car crash in a racing game involving a large number of vehicles would cause a major spike in processing time used for collision detection and response. In such transient overload situations, it would be advantageous if the processing load of the AI characters can be throttled up or down to respond to these dynamic changes.

4. Effective load management technique for AI characters

Figure 1 shows an overview of the load management technique. The AI LOD (Level of Detail) system provide a LOD value for each of the AI characters that indicates the importance of the AI characters. Using the LOD values and the current CPU utilization of the game, the scheduler calculates the optimal execution periods of the AI characters. Execution of the tasks (AI characters' updates) is the responsibility of the task manager. In addition, the task manager will ensure that the time budget per frame is not exceeded by delaying the update of the AI characters if the processing time remaining is inadequate.

4.1. The elastic task model

An elastic model is introduced by Buttazzo et al. where tasks are modeled as a spring system where increasing or decreas-

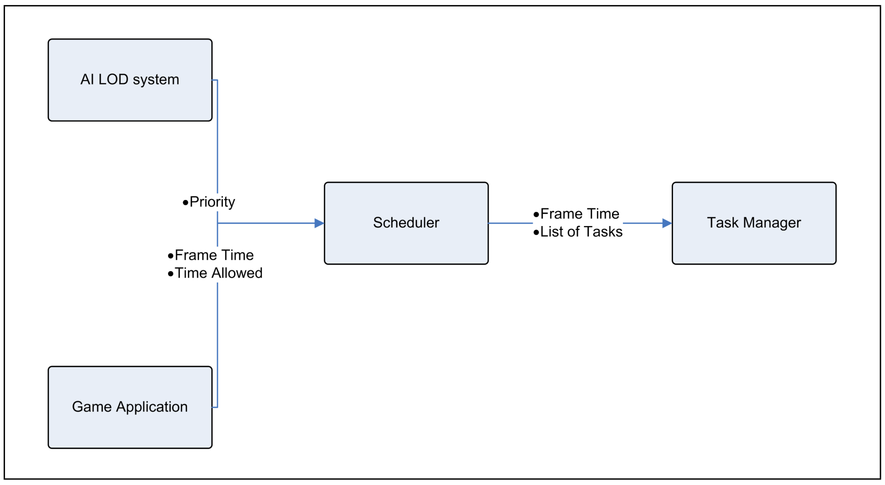


Figure 1: *Overview of the proposed technique.*

ing a task period is analogous to the compression or decompression of a spring [BLCA02]. Each task τ_i is characterized by five parameters:

$$\tau_i(C_i, T_i, T_{io}, T_{imax}, E_i) \text{ for } i = 1, \dots, N \tag{1}$$

where N is the number of tasks in the system. Computation time C_i is the execution time of task τ_i. The nominal period T_{io} and maximum period T_{imax} denote the minimum and maximum boundaries of the task period T_i. The most desirable output is perceived when the task is updated at the nominal period T_{io}. The elasticity coeffecient E_i represents the resistance of the task τ_i to increasing its period. The larger the value of E_i, the more elastic the task is thus the bigger the increase in its execution period when system workload increases.

The CPU utilization factor of the periodic task is defined as:

$$U_i = \frac{C_i}{T_i} \tag{2}$$

and the total utilization of the sytem of n processing tasks as:

$$U_p = \sum_{i=1}^{n} U_i \tag{3}$$

The elastic task model can be used for AI characters because the updates of AI characters are periodic in nature. Every AI characters have to retrieve information from the game environment (Sense), process this information (Think) and produce a reaction (React) at a certain rate [vLLB*99, Nar02]. The rate of update can be different for each of the AI characters and degradation of quality when the execution frequencies are reduced would not introduce critical errors into the game. AI characters may become unresponsive or display incorrect behaviours but these problems do not change the outcome of the game itself.

The computation time C_i of each AI character's behavioural updates are tracked in real-time to provide an accurate measure of how much computational power is used by the AI characters. The LOD value of the AI characters is used as the elastic coefficient factor for the AI characters. For the purpose of discussion in this paper, the term *task* would be defined as the periodic update function of each AI character in a game.

4.2. AI level of detail (LOD)

LOD algorithms in AI are similar to the LOD used in graphics programming. The common concept here is to allocate more computing time to AI characters that are most important from the player's perspective [Mil06]. An effective LOD algorithm can be provided simply by varying the update frequency of each AI characters based on its importance. Important characters can received more processing time by being scheduled to update more frequently. This is the basis of the *Scheduling LOD* [Mil06].

In our approach, the distance of the AI character to the camera is utilized as the elasticity coefficient factor of the AI character. This also corresponds to the importance of the AI characters and subsequently the LOD as well. The larger the distance, the less important the AI character. This method is lightweight enough to allow the value to be recalculated every frame.

Unfortunately, this straightforward approach may not be suitable for certain scenarios. Additional parameters such as the visibility of the AI characters may need to be considered [NG05]. In cases where the camera distance and visibility tests are not adequate, the LOD value can be further fine-tuned by taking into considerations of game specifics information such as:

a) **Type**. Different techniques may be used to implement the behaviours of the different types of AI characters. If the underlying technique is vulnerable to produce low quality result when executing at low frequencies, it would help if these AI characters are given higher priorities.

b) **Size**. The details of a larger AI character are more visible to the human player. In contrast, a smaller AI character may not be as obvious. Therefore, comparatively smaller sized AI characters can be allocated less computation time.

c) **Interactivity**. If the AI characters are meant to be highly interactive and responsive to the human player, then these characters should be allocated higher update frequencies to ensure that it respond to the human player in a timely manner.

The task of providing this heuristic depends heavily on the game specifics and context. Therefore, the derivation of the LOD value should be customized to suit the game if necessary.

4.3. Calculating the task periods

The scheduler receives three input parameters: the list of tasks Γ, processing time allowed per frame T_d and the frame time (total time taken for processing since the previous game frame) T_f . The desired utilization factor that the AI characters are allowed to consume is defined as:

$$U_d = T_d/T_f \tag{4}$$

The total nominal utilization U_o is defined as the total sum of the tasks' nominal utilization factors and the sum of task elasticity coefficient factors E_v is initialized to the total sum of the elasticity coefficient factors of all tasks.

$$U_0 = \sum_{i=1}^{N} C_i/T_{io} \tag{5}$$

$$E_v = \sum_{i=1}^{N} E_i \tag{6}$$

Using the analogy of a spring system, we assume that all springs are in the uncompressed state initially and the compression is performed on the spring system from this initial state of rest. When applied to the AI tasks, the utilization factors of all tasks are set at their nominal values before the elastic compression algorithm is performed. Therefore, the total utilization factor of fully compressed tasks U_f is set to zero and the total nominal utilization factor of the set of tasks undergoing compression U_{vo} is initialize as U_0.

This modification differentiates our technique from the original algorithm. Buttazzo et al.'s algorithm is more suitable for fixed priority scheduling where the coefficient factors are pre-assigned and remains static [BLCA02]. In constant overload conditions, the task periods of the set of fully-compressed tasks would remain unchanged at their maximum values T_{imax}. This would result in priority inversion, where AI characters of growing importance are not scheduled to update at shorter time intervals because of the amount of compression applied to the corresponding AI task. Our modification allow AI characters of higher importance to be re-assigned smaller periods regardless of the amount of compression applied to the tasks. This maintains the level of details (LOD) effect for the different AI characters.

Another major difference is when a feasible schedule cannot be found, new incoming tasks are not rejected. When the scheduler is unable to meet the desired utilization factor, CPU utilization factors of all AI tasks would be fully compressed to their minimum values. We expect such conditions to be rare and transient in nature, therefore the best effort approach utilized here would be preferable to rejecting the addition of new AI characters into the game world.

The algorithm to calculate the task periods for the functional updates of AI characters is shown in Figure 2.

4.4. Task execution and limiting processing time per frame

In a conventional game loop, the update frequencies of all game components are the same as the frame rate of the game. Therefore, we decoupled the simulation time from the actual real time by using an approach similar to the concept of "virtual time" [HM02]. To the task manager, simulation time T_{simul} is just a numeric value that can compared, manipulated and computed independently of the actual real time. At the end of every game frame, the time elapsed which is the frame time T_f is calculated and used to advance the simulation time T_{simul}. This allows independent update frequencies for each AI task as well as finer control of the task periods.

Each task τ_i maintains 3 additional execution parameters: the next simulation time that the task is due to execute T_{nexti}, the last simulation time that the task has executed T_{lasti} and the number of time the execution of the task has been delayed by the task manager M_i. The elapsed time perceived by each task τ_i is simply the value of $T_{nexti} - T_{lasti}$.

The task model as seen from the perspective of the task manager:

$$\tau_i(C_i, T_i, T_{nexti}, T_{lasti}, M_i) \text{ for } i = 1, \ldots, N \tag{7}$$

In every frame, the task manager determines the list of tasks due for execution Γ_{frame} by comparing the next execution time T_{nexti} with the current simulation time T_{simul}. Then

```
**ElasticTaskCompression** ($\Gamma$, $T_d$, $T_f$)

$U_d = T_d/T_f$
$U_o = \sum_{i=1}^{N} C_i/T_{io}$
InitialState = 1
Do {
    $U_f = E_v = 0$
    For each $\tau_i$ in $\Gamma$ {
        If ($T_i < T_{imax}$ Or InitialState == 1)
            $E_v = E_v + E_i$
        Else
            $U_f = U_f + U_i$
    }
    ok = 1
    If ($E_v > 0$) {
        $U_{vo} = U_o - U_f$
        For each $\tau_i$ in $\Gamma$ {
            If ($T_i < T_{imax}$ Or InitialState == 1) {
                $U_i = U_{io} - (U_{vo} - U_d + U_f)(E_i/E_v)$
                If ($T_i > T_{imax}$) {
                    $T_i = T_{imax}$
                    ok = 0
                }
                Else
                    $T_i = T_{imax}$
            }
        }
    }
    InitialState = 0
} While (ok == 0)
```

Figure 2: *Algorithm for calculating the task periods.*

the list of tasks is sorted by M_i and then by the time elapsed $(T_{nexti} - T_{lasti})$. Tasks which has been delayed the highest number of times (largest M_i) will execute first followed by tasks with the largest time elapsed. Ordering the list of tasks in this manner ensures that delayed tasks are given priority for execution and prevent starvation of lower priority tasks with larger execution period when the processing time budget T_d is not adequate for the processing requirements of the all the tasks scheduled for the current frame.

The algorithm to manage the execution of the tasks is shown in Figure 3.

5. Experiments and results

The prototype is implemented using a proprietary commercial game engine which runs on the Microsoft Xbox 360 hardware. The game engine is configured to run a test simulation of a typical North American desert environment. The environment includes four different types of wildlife animats; wild hares, javelinas, groundhogs and bobcats with 25 animats for each type making a total of 100 animats. State machines containing the behaviours and animation for each wildlife type are defined using scripts. These forms the bulk of the processing contained within the update function of the wildlife animats. Therefore, when we apply the load management technique to these 100 wildlife animats to manage the execution of their update function we noticed changes in the animation quality of the wildlife animats as quality is traded for performance and vice-versa by the technique. Figure 4 shows a screenshot of the simulation.

```
**ManageTaskExecution** ($\Gamma$, $T_d$, $T_f$, $T_{simul}$)

$T_{simul} = T_{simul} + T_f$ $\Gamma_{frame} = emptyset$
For each $\tau_i$ in $\Gamma$ {
    If ($T_{nexti} \leq T_{simul}$)
        $\Gamma_{frame} = \Gamma_{frame} + \tau_i$ }
Sort tasks in $\Gamma_{frame}$ by $M_i$ descending
and by ($T_{nexti} - T_{lasti}$ descending if $M_i$ are equal)
For each $\tau_i$ in $\Gamma_{frame}$ {
    If ($C_i \leq T_d$) {
        Execute task $\tau_i$
        $T_d = T_d + C_i$
    }
    Else {
        $M_i = M_i + 1$
        $T_{nexti} = T_{simul}$
    }
}
```

Figure 3: *Algorithm for managing the execution of tasks in every frame.*

Figure 4: *Simulation of 100 wildlife animats*

All the tasks (wildlife animats) are assigned the scheduling parameters displayed in Table 1.

Two series of experiments were performed on the simu-

Parameter Name	Value
Nominal Task Period T_{io}	33.33 ms
Maximum Task Period T_{imax}	80.0 ms
Elastic Coefficient Factor E_i	1
Computation Time C_i	0.0 ms

Table 1: *Scheduling Parameters*

lation. First, we investigate the effect of assigning different processing time budget per frame T_d. Second, an extra processing load is induced to increase the frame time T_f . We observed and study the reaction of the simulation while under the influence of the load management technique.

5.1. Load management under different time constraints

The best quality in animation is perceived when all the animats are executing at their nominal task periods which is 33.33 ms. The elastic scheduling would try to minimize the pertubations of the task periods from their nominal values in order to maximize the quality of service. Therefore, a measure of the quality of service shown by the simulation can be formulated as the deviation of the task periods from their nominal values. The absolute deviation of the task periods from its nominal values for N number of tasks is defined as:

$$D_i = T_i - T_{io} \tag{8}$$

$$D_{period} = \frac{1}{N}\sum_{i=1}^{N} D_i \tag{9}$$

The performance of the simulation under three different processing time constraints (per frame): 10.0 ms, 20.0 ms and 30.0 ms is shown in Table 2.

T_d (ms)	C_{total} (ms)	U_{total}	D_{period}	FPS	Overhead (ms)
10.0	10.07	0.4618	47.9289	44.64	0.1194
20.0	16.745	0.5507	36.9175	32.69	0.2139
30.0	28.155	0.7514	16.2765	26.73	0.1616

Table 2: *Performance evaluation of the technique when subjected to different processing time constraints. Data shown are average values*

The results shows that the total computation time of the simulation is restrained to comply with the user-specified limits. This contributes to a higher and more stable frame rate in the game. The proposed technique is also very efficient and consumes minimal overheads (approximately 0.2 ms or less).

However, reducing the processing time budget per frame T_d comes at the cost of higher deviation of the task periods D_{period}. This correlates with degradations in the quality of the wildlife animats' animation during visual inspection of the simulation.

5.2. Response to fluctuations in processing time (frame rate)

In this experiment, the processing time budget is set at 30.0 ms per frame and an additional load is induced from sample frame 200 to sample frame 400, to increase the time taken to complete iterations of the game loop. This allows us to investigate the response of the load management technique when subjected to an external load.

The graphs in Figure 5(a) shows that the total computation time C_{total} of the wildlife animats remains fairly stable at 30.0 ms per frame even when the additional load is induced. This is achieved by reducing the total utilization factor U_{total} of the wildlife animats for the duration of the additional system load (see Figure 5(b)). Such mechanism is useful to reduce the impact on the controlled module from other game components' workload.

6. Limitations and future work

Our proposed technique assumes that increasing the time interval between executions of the AI characterŠs update function would not result in an increase of the computation time required. If this is not true, the overload situation may worsen as the processing time of each compressed task increases. This is possible when the underlying AI technique breaks the time elapsed into smaller time steps to be fed into a simulation loop. Larger time intervals would result in more time steps being generated and subsequently longer time spent in the simulation loop. In such scenarios, the technique becomes counterproductive to the effort of reducing the CPU utilization of the AI characters. We are currently investigating measures that can be used to mitigate this effect.

Finally, we are working to extend the technique to create a complete framework to manage the computational requirements of game AI. The final framework would be compatible and increase the effectiveness of optimization techniques that target the actual AI algorithm such as Anytime Algorithms [GZ95,Gra96]. Similar to Wright and Marshall's effort in creating a general-purpose framework for game AI processing, advanced capabilities like different execution methods and minimising the number of tasks scheduled for each individual frames [WM00] would be part of this framework that we are creating.

7. Conclusions

We have presented an efficient and effective technique to manage the computational load of AI characters in games.

By using our technique, AI processing becomes more resilient to transient changes in game workload and the processing time required per frame can be constrained according to user specifications. This results in better predictability and stability in the runtime performance of games. The implementation of our technique is designed to be nonintrusive and compatible with the architectures of existing games, making it easier for game developers to integrate and use.

We believe that the technique presented serves as a valuable tool for game developers to manage the computational load of AI characters in games. Although the need for proper profiling and manual optimizations are not totally removed, the effort required to maintain the delicate balance of performance and quality is significantly lower.

References

[Ale02] ALEXANDER B.: An architecture based on load balancing. *AI Game Programming Wisdom* (2002), 298–304.

[BA02] BUTTAZZO G., ABENI L.: Adaptive workload management through elastic scheduling. *Real-Time Syst. 23*, 1-2 (2002), 7–24.

[BCKW98] BONAKDARIAN E., CREMER J., KEARNEY J., WILLEMSEN P.: Generation of ambient traffic for real-time driving simulation. In *Proceedings of 1998 Image Conference* (1998).

[BLCA02] BUTTAZZO G., LIPARI G., CACCAMO M., ABENI L.: Elastic scheduling for flexible workload management. *Computers, IEEE Transactions on 51*, 3 (2002), 289–302.

[Bro02] BROCKINGTON M.: Level-of-detail ai for a large role-playing game. *AI Game Programming Wisdom* (2002), 419–425.

[CHL06] CHANTEM T., HU X. S., LEMMON M.: Generalized elastic scheduling. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International* (2006), 236–245.

[DHOO05] DOBBYN S., HAMILL J., O'CONOR K., O'SULLIVAN C.: Geopostors: a real-time geometry / impostor crowd rendering system. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2005), ACM, pp. 95–102.

[dSM06] DA SILVEIRA L. G., MUSSE S. R.: Real-time generation of populated virtual cities. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2006), ACM, pp. 155–164.

[Gra96] GRASS J.: Reasoning about computational resource allocation. *Crossroads 3*, 1 (1996), 16–20.

[GZ95] GRASS J., ZILBERSTEIN S.: *Anytime Algorithm Development Tools*. Tech. rep., Amherst, MA, USA, 1995.

[HM02] HARVEY M., MARSHALL C. S.: Scheduling game events. *Game Programming Gems 3* (2002), 5–14.

[KDC*08] KAVAN L., DOBBYN S., COLLINS S., ŽÁRA J., O'SULLIVAN C.: Polypostors: 2d polygonal impostors for 3d crowds. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 149–155.

[McL02] MCLEAN A. W.: An efficient ai architecture using prioritized task categories. *AI Game Programming Wisdom* (2002), 290–297.

[Mil06] MILLINGTON I.: *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[MUAT05] MUSSE S. R., ULICNY B., AUBEL A., THALMANN D.: Groups and crowd simulation. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 2.

[Nar02] NAREYEK A.: Review: Intelligent agents for computer games. In *CG '00: Revised Papers from the Second International Conference on Computers and Games* (London, UK, 2002), Springer-Verlag, pp. 414–422.

[NG05] NIEDERBERGER C., GROSS M.: Level-of-detail for cognitive real-time characters. *The Visual Computer 21*, 3 (4 2005), 188–202.

[RD05] RYDER G., DAY A. M.: Survey of real-time rendering techniques for crowds. *Computer Graphics Forum 24*, 2 (2005), 203–215.

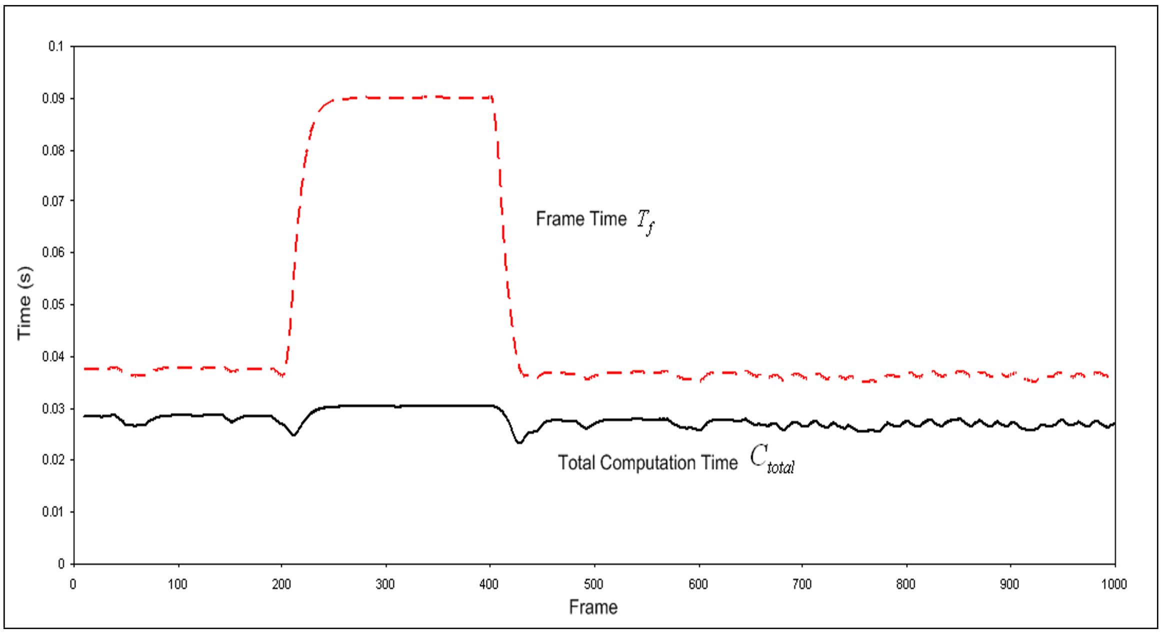
[Rey87] REYNOLDS C. W.: Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM, pp. 25–34.

[Rey06] REYNOLDS C.: Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), ACM, pp. 113–121.

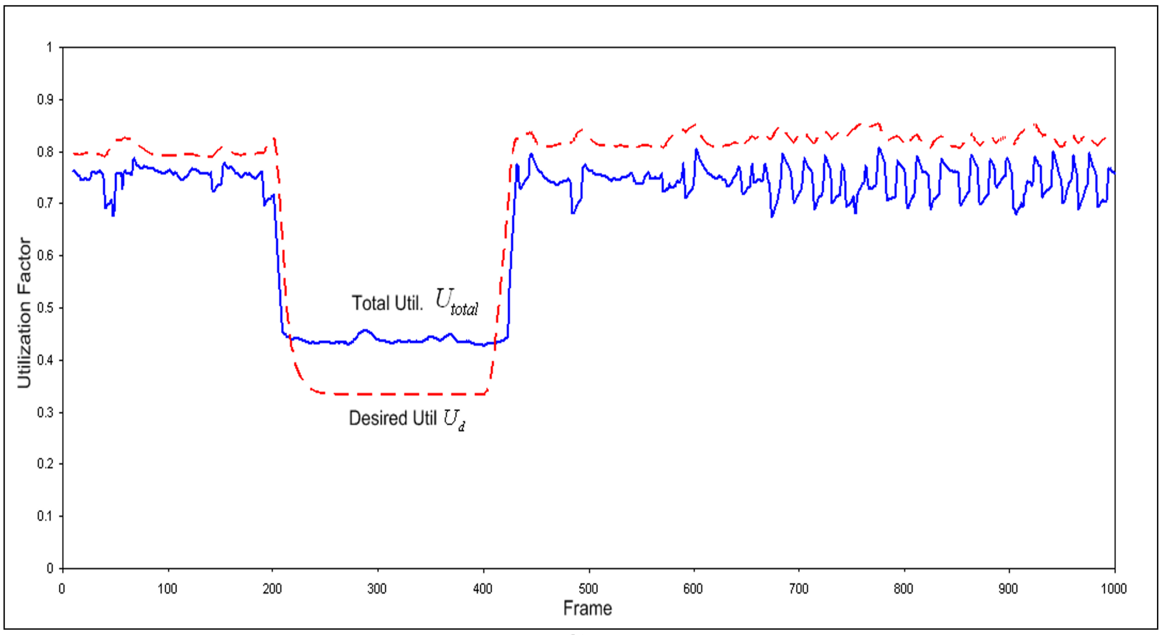
[TCP06] TREUILLE A., COOPER S., POPOVIĆ Z.: Continuum crowds. *ACM Trans. Graph. 25*, 3 (2006), 1160–1168.

[vLLB*99] VAN LENT M., LAIRD J., BUCKMAN J., HARTFORD J., HOUCHARD S., STEINKRAUS K., TEDRAKE R.: Intelligent agents in computer games. In *AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence* (Menlo Park, CA, USA, 1999), American Association for Artificial Intelligence, pp. 929–930.

[WM00] WRIGHT I., MARSHALL J.: Egocentric ai processing for computer entertainment: A real-time process manager for games. *Game-On 2000, 1st International*



(a)



(b)

Figure 5: *(a) Processing time consumed by the wildlife animats and the game frame time. (b) Changes in the CPU utilization.*

Conference on Intelligent Games and Simulation (2000), 42–46.

[Woo00] WOODCOCK S.: Game ai: The state of the industry. *Game Developer* (2000).