Multi-threaded Game Engine Design

James Tulip
Charles Sturt University
School of Information Technology
Panorama Av, Bathurst, 2795
+61 2 63384931
jtulip@csu.edu.au

James Bekkema
Charles Sturt University
School of Information Technology
Panorama Av, Bathurst, 2795
+61 2 63384724
jbekkema@csu.edu.au

Keith Nesbitt
Charles Sturt University
School of Information Technology
Panorama Av, Bathurst, 2795
+61 2 63384262
knesbitt@csu.edu.au

ABSTRACT

Game engines are specialized middleware which facilitate rapid game development. Until now they have been highly optimized to extract maximum performance from single processor hardware. In the last couple of years improvements in single processor hardware have approached physical limits and performance gains have slowed to become incremental. As a consequence, improvements in game engine performance have also become incremental. Currently, hardware manufacturers are shifting to dual and multi-core processor architectures, and the latest game consoles also feature multiple processors. This presents a challenge to game engine developers because of the unfamiliarity and complexity of concurrent programming. The next generation of game engines must address the issues of concurrency if they are to take advantage of the new hardware. This paper discusses the issues, approaches, and tradeoffs that need to be considered in the design of a multi-threaded game engine.

Categories and Subject Descriptors

C.4.1 [**Computer Systems Organization**]: Performance of Systems – *design studies.*

General Terms

Algorithms, Performance, Design

Keywords

Game Engine, Multi-Threaded

1. INTRODUCTION

Computer gaming is a vast industry, rivaling Hollywood and the music industry in terms of revenue. The global market is estimated to be worth over 40 billion USD in 2006 and is one of the most rapidly expanding sectors of the global economy. Game consoles have significant mass-market penetration, being present in over three quarters of households in the US and Australia [4][7]. The industry is growing rapidly and computer games are becoming ever more visually appealing, sophisticated and expensive to develop. Major console vendors such as Sony, Microsoft and Nintendo are even now releasing new more powerful game consoles supporting even more graphically intense games with even more content. Increasingly, game technology is also being applied in non-game fields such as military and medical training [9]. These non-game (or 'serious game') applications often place high demands on the verisimilitude of the physical simulation and AI aspect of the application, as well as the graphical representation of the situation being simulated.

Paradoxically, as the complexity, content and expense of games increases, the time to market and shelf life of a typical game is decreasing. Game development is a highly risky business with only one in ten games becoming a hit (or even repaying development costs). Much of the cost of game development is incurred implementing highly optimized but non game specific functionality, such as rendering, scene management, and physical simulation. Increasingly, game developers have turned to specialist middleware to shorten development times, and reduce risk. Instances of such specialized middleware are referred to as game engines. Game engines provide reliable, well tested, and highly optimized generic game functionality. The high cost of developing such middleware is amortized over many games.

Over the last decade, exponential improvements have occurred in processor speed, the amount of memory available, the size and speed of external storage, and the speed and quality of graphic hardware, not to mention sound capabilities. More recently however, the relentless doubling of processor speed every year and a half has tapered off as the physical limits of current silicon chip technology have been approached. The major chip manufacturers (Intel and AMD) have turned to dual and multi-core processor architectures in an effort to keep pace with the demand for ever increasing processor power [11]. The new generation of game consoles from Sony (PS3), and Microsoft (Xbox 360) also feature multi-core architectures.

Game engines have traditionally sought to extract maximum performance out of their implementation platform. As a result, existing game engines are highly optimized to run efficiently on single processor architectures and have eschewed multi-threaded designs. This is due to the performance overhead of threading on single processor architectures, as well as the perceived complexity and non-determinism of threaded programs. However, the next generation of game engines will need to address the complexities of concurrent programming (that is multi-threaded programming) if they are to extract the maximum performance out of the new PC and game console platforms. Ultimately, a game engine that leverages multi-processor hardware must be designed from the beginning with concurrency as a key feature [1][5].

Few existing game engines have been designed to take advantage of multi-processor and multi-core architecture. This is illustrated by a performance test carried out with three current games, Serious Sam2, Quake 4, and FarCry. The three games were run on both a single core, Athlon 64 3800+ computer and a dual core,

Athlon 64 X2 3800+ machine. Performance was measured in frames per second (FPS) and for all three games a decrease in performance of between 14-18% was found when running on a dual core compared to the single core architecture (see Table 1).

Table 1. The overheads of running on dual-core hardware result in a decrease in performance with three current games.

Computer Game	Performance (FPS)		
	Single-core processor Athlon 64 3800+	Dual-core processor Athlon 64 X2 3800+	Decrease in peformance with dual-core
Serious Sam 2	106	93	-13 (14%)
Quake 4	99	86	- 13 (15%)
Far Cry	75	67	- 8 (18%)

Little has been published on the application of concurrent programming techniques to game engine architecture because of the historical focus on extracting maximum performance from single processor architectures. This paper seeks to address that situation by discussing the issues, approaches and tradeoffs that need to be considered in the design of a multi-threaded game engine.

The paper is organized as follows: Section 2 reviews some basic concurrent programming theory. Section 3 reviews game engine architecture, in particular, the functional decomposition, and data flow dependencies between components. Section 4 looks at opportunities for parallelism in game engines. Section 5 synthesizes the preceding sections by discussing possible approaches to multithreaded game engine design and draws some conclusions on the optimum approach to take.

2. CONCURRENT PROGRAMMING

2.1 Amdahl's Law

Amdahl's Law states the fundamental limit to potential speed gain due to concurrent processing. Essentially it states that the speed gain is proportional to the number of processors executing the concurrent part of the code but inversely proportional to the amount of sequential code. More formally and accurately:

$$S = \frac{1}{(F + (1-F)/N)}$$

where S is the speedup due to using N processors, and F is the fraction of sequential code.

Amdahl's Law has been interpreted as 'the law of diminishing returns' whereby increasing the number of processors has a diminishing effect, even on highly parallel code.

However, the main implication of Amdahl's Law for a given number of processors is that the maximum speedup due to concurrency is limited by the amount of irreducibly sequential code contained in an algorithm. Even a small amount of sequential processing limits the maximum speedup so that a large fraction of the maximum speedup is gained with a low number of processors [5]. The greater the fraction of sequential processing, the fewer processors are required to attain most of the speedup (See Table 2).

Table 2. Potential speedup due to processor number and fraction of sequential code in algorithm.

Fraction of Sequential Code (F)	Speedup due to number of processors (N)				Maximum Potential Speedup (S) (infinite N)
	2	4	8	16	
0.9	1.05	1.08	1.11	1.11	**1.11**
0.8	1.11	1.18	1.21	1.23	**1.25**
0.7	1.18	1.29	1.36	1.39	**1.43**
0.6	1.25	1.43	1.54	1.60	**1.67**
0.5	1.33	1.60	1.78	1.88	**2.00**
0.4	1.43	1.82	2.11	2.29	**2.50**
0.3	1.54	2.11	2.58	2.91	**3.33**
0.2	1.67	2.50	3.33	4.00	**5.00**
0.1	1.81	3.08	4.71	6.40	**10.00**
0.01	1.98	3.88	7.48	13.91	**100.00**
0.001	2.00	3.99	7.94	15.76	**1000.00**

2.2 Task and Data Parallelism

Task and data parallelism refer to two major approaches to concurrency. In task parallelism, the overall processing is split into a number of separate tasks which execute independently and asynchronously. Execution of these tasks may be coordinated by some well defined synchronization points, where data may be exchanged. In data parallelism, the data is broken up into independent chunks, and the same processing is applied independently to the different chunks in parallel [3].

Task parallelism is extremely useful for handling high latency operations such as file IO, or network communications. It allows useful processing to continue while blocking IO operations are dealt with in separate threads. It is also useful in constructing a 'cascade' or pipeline of processing where a large amount of data is subject to a sequence of dependent processing steps. Task parallelism is an extremely flexible approach that gives a programmer a lot of control over the number of threads executing, and their coordination. On the other hand, synchronization issues can make task parallelism difficult to implement efficiently.

Data parallelism is useful when the same operation must be applied to a large amount of data, and there are no dependencies between the operations applied to different parts of the data.

2.3 Synchronization

The word 'synchronized' in the context of concurrency is used in the sense of 'coordinated'. In order to ensure that data does not become corrupted, mutual exclusion to shared resources must be enforced by the acquisition of locks [6]. Code between when a lock is acquired and when it is released is referred to as a 'critical

region' and may only be executed by a single thread at any one time. Enforcement of serial access to such critical regions is referred to as synchronization. Clearly the amount of code executed with such critical regions adds to the amount of irreducibly sequential code within a program and hence limits the maximum speedup possible. There is also additional processing overhead in acquiring and releasing locks.

Synchronization may also refer to the coordination of data parallel processing by the creation of 'barriers'. Barriers are used to ensure that all data processing reaches a certain stage, before a subsequent phase of processing is carried out.

Any use of locks also introduces the possibility of creating deadlocks which result from circular lock dependencies between concurrent processes. These dependencies can be subtle, indirect, and timing dependent. The obscure bugs caused by interacting dependencies are a major reason why concurrent programming is regarded as arcane, and unreliable.

2.4 Granularity

Amdahl's Law assumes no overhead for switching between concurrent tasks. In fact these overheads can be large. The key concept in this regard is granularity, which refers to the amount of work executed per concurrent task. A fine grained breakdown of tasks results in relatively higher levels of system overhead since each task performs only a small amount of work while the overhead of managing each thread remains constant regardless of thread size. There is also memory overhead associated with each concurrent process in the form of a local stack and possibly local heap. These overheads combine to use up system resources and make fine grained threading approaches relatively inefficient in terms of useful application work performed compared to system overheads [2].

2.5 Load Balancing

Load balancing refers to the concept that the processing load allocated to tasks executing in parallel should be roughly similar. This concept in some ways works in opposition to the concept of granularity. A fine grained division of processing load is likely to result in there always being some task that can be executed, keeping all processors busy. A coarse granularity may result in some tasks containing markedly more processing between synchronization points than others. If some tasks take markedly longer to execute than others, poor utilization of processor resources in the system may result [5].

3. GAME ENGINE ARCHITECTURE

Game engines typically are responsible for all the non-game specific functionality of a game application. A game application consists of much more than the core gameplay and game world usually thought of as the game [8]. The core game is embedded in a framework that provides services such as input control configuration, save/load functions, multiplayer network game setup, graphic and sound configuration, resource loading, and even game parameter adjustment such as the degree of difficulty, and number of opponents. The game framework facilitates switching between different modes of interacting with the application. The actual game is but one of these modes, albeit the most important.

At the core of the actual game is a programming structure known as the game loop. The game loop consists of a sequence of tasks necessary to implement a real time interactive simulation. This sequence of tasks must be executed between displaying successive frames to the screen, typically in around 33ms.

Before entering the game loop, there is typically an initialization phase where the bulk of any necessary resources are loaded. Resources may also be loaded on demand as a result of actions within the game loop.

The first task within the game loop is gathering input. This process must be nearly instantaneous in order to give the user a feeling of quick response. Input is gathered either by polling the state of input devices, or accumulating input device state changes in a buffer by responding to input hardware events. Hardware events are translated into game specific and meaningful game actions which are enacted in the subsequent update step.

Network messages can be regarded as a specialized form of input, but they represent a more highly processed form of game message than basic game actions and are processed independently.

The next task is to run the game simulation. This includes generating AI behaviours, running the physical simulation, updating any particle systems, and running the game logic. The player and camera position and orientation are also updated. This 'update' step is divided into two phases. The first phase uses input and the current game state to calculate a proposed new game state. In the second phase collisions and game entity interactions are detected and resolved. The update step is applied to representations of game entities used within the game world rather than to renderable graphics objects.

The final task is to render the updated game world to screen and speaker. The visible and audible part of the game world is calculated, and animation transforms, including interpolations between animation sets and keyframes, are applied to the renderable representations of game entities. Lighting and texture states are calculated, and the sound field is generated. The outputs of this step are data structures that can be used to deliver renderable data to the graphics and sound hardware in the most efficient manner.

The render step may be decoupled from the game world update step, and the update step run at a lower frequency than the render step. When this occurs, interpolation and extrapolation of game entity positions and animations occurs in the render step without reference to the physical simulation between game state updates.

The functional decomposition of games outlined above has lead to a fairly standard modularization of game engine functionality. There is a 'kernel' which handles scheduling of tasks within the game loop. There is a resource management sub-system, an input sub-system, a network sub-system, a sound sub-system, and a rendering sub-system. Commonly, there is a scene management sub-system used in the rendering step, and a physics sub-system used in the update step for collision detection and response. A GUI sub-system may be provided for use in the game framework and in-game user interface. Other specialized sub-systems may also be provided, such as for particles, water and weather simulation. Low level AI functionality such as path-finding may also be provided as part of a game engine's function, but in

general, game entity behaviours and game logic are too game specific to be provided by a generic game engine.

4. OPPORTUNITIES FOR PARALELLISM IN GAME ENGINES

The standard modularization of game engine functionality has resulted in a breakdown into kernel (scheduler), resource management, input, network, physics, AI, scene management, sound, and render modules. Some of these are obvious candidates for task parallelism.

Resource loading is a prime candidate for task parallelism since it is a high latency blocking IO operation. Some current game engines (for example Auran Jet) already use a separate thread for this task. The results of a resource loading request can be checked once per update cycle.

Input and network message processing are also good candidates for task parallelism since they too are high latency IO operations. These tasks are pure producers of information and there are no dependencies on any other part of the engine in the production of that information. The interaction of these tasks with the rest of the engine can be encapsulated as output buffers containing records which the rest of the engine consumes.

On the other hand, sound and render modules operate purely as consumers of information. The results of their operation are serially output to hardware and displayed to the user, but there are no programmatic dependencies on their operation. Interestingly, if several frames are rendered between game world updates, there are no dependencies between frames, only on the current game world state. This implies that all frames between game world updates could be rendered in parallel.

Scene management is a buffering step between the game-world update phase and the rendering phase whose function is to limit the amount of information the graphical and audio rendering tasks have to process. Scene management has no impact on the game world, and is purely a preprocessing step which tailors game world information to the particular view required for rendering. Visibility and audibility testing of individual units or nodes within a scenegraph is a data parallel operation.

Following scene management, rendering operations are entirely data parallel, up to the point where the final results must be combined and sent to the rendering hardware in an optimized serial manner. Data parallel graphical operations include all interpolation between animation keyframes, ‘skinning’ of models, and application of lighting and textures. Data parallel sound operations include calculation of individual sound source contributions to the overall perceived sound.

Building the final render tree, which involves sorting of graphics primitives on the basis of textures, lighting, and other effects to allow efficient use of graphics hardware is an intrinsically serial operation.

In between these producers and consumers of information are the physics, AI, and non-engine game logic, which both consume and produce within-game information. Somewhat counter-intuitively, the central game-world processing phase has no inherent external dependencies. The game world simulation can and must execute without necessarily receiving any user input, or network messages. Similarly, the simulation can execute regardless of whether the scene is rendered or any sound produced.

Inside the game-world processing phase there is one synchronization point which cannot be avoided. There is a point where interactions between game world entities must be resolved. Not only must first-order interactions be resolved, but new interactions generated as a consequence of resolving those first-order interactions must also be resolved, and so on in a recursive manner until there are no new interactions generated by the resolution of old ones. Furthermore, the new interactions may involve entities not previously involved in interactions during the current resolution cascade. This interaction resolution cascade is intrinsically serial.

However, the initial phase of applying physics and AI to generate first-order positions, trajectories, and interactions, depends purely on the previous states of game world entities and whatever inputs happen to have occurred during the previous update cycle. So the initial phase of the update cycle is intrinsically data parallel. Even during the interaction resolution phase there are many separate interactions that don’t necessarily overlap so there are also opportunities for parallelism in the resolution phase. However, these opportunities might be difficult to detect and separate efficiently.

The three major phases of the game loop are in fact largely independent tasks, operating in a classic producer/consumer pattern and suitable for task parallelism. The input and render phases themselves are comprised of separate sub-tasks also suitable for task parallelism. Within the update and rendering phases there are also significant opportunities for data parallelism. However, at the very core of the update phase is an intrinsically serial interaction resolution cascade [10]. Mechanisms for improving the concurrency of this resolution cascade will form the basis of much future research.

5. DISCUSSION

There are a number of guiding principles that emerge from the review of concurrency. First of all the number of threads should be minimized, consistent with taking advantage of the number of processors available. Secondly, creation and destruction of threads during processing should be avoided. These principles stem from the desire to minimize system overhead due to thread management and context switching. Thirdly, synchronization should be minimized and fourthly the workload assigned to processors should be as balanced as possible.

Three of the high level tasks identified (input, network, and resource loading) exhibit I/O blocking behavior. It is probably desirable to have these tasks handled by dedicated threads since such I/O blocking threads are low overhead long lived service threads that encapsulate high latency operating system interactions.

However, the update and render tasks are another matter. Both update and render involve a large amount of data parallelism, but also involve sequences of operations with rigid processing order requirements. If this data parallelism is exploited by creating and destroying many threads, it would create a lot of system overhead without any performance benefit given the number of processors

available on current and foreseeable systems. These issues may be avoided through the use of a thread pool.

A thread pool is a collection of reusable 'worker' threads which pick up executable tasks from a 'task pool' and process them. Thread pools allow the number of possible threads in a system to be controlled, preventing resource starvation. They eliminate the overhead of thread creation and destruction and decouple the work performed from the executing process. Also, by allowing tasks to be executed by any available worker thread, they encourage load balancing between processors. Synchronization costs may be avoided by framing tasks as the non-blocking processing required between synchronization points. However, in order to accomplish the original purpose the entire process, tasks before and after the synchronization point must be scheduled correctly, introducing the idea of priority queuing to the task pool. Non-blocking processing of tasks also requires that all tasks have non-blocking access to their output buffers.

Input, network monitoring, and resource loading can be run as separate high priority threads, outside the game loop. These threads would mainly be blocked or idle. The initial phase of the update step is strongly data parallel and can be handled using a threadpool to process many small AI and physics simulation tasks. Rendering is also a largely data parallel phase that may also best be handled with a threadpool processing a prioritized task queue.

In order to test these concepts, we have developed a prototype based on the concept of a 'task tree' serviced by a threadpool. Figure 1 shows the breakdown of game loop processing tasks and the scheduling order we have used in our prototype. Our prototype does not include high level tasks such as network interaction and resource loading. This is because such tasks do not necessarily complete during a single cycle of the gameloop.

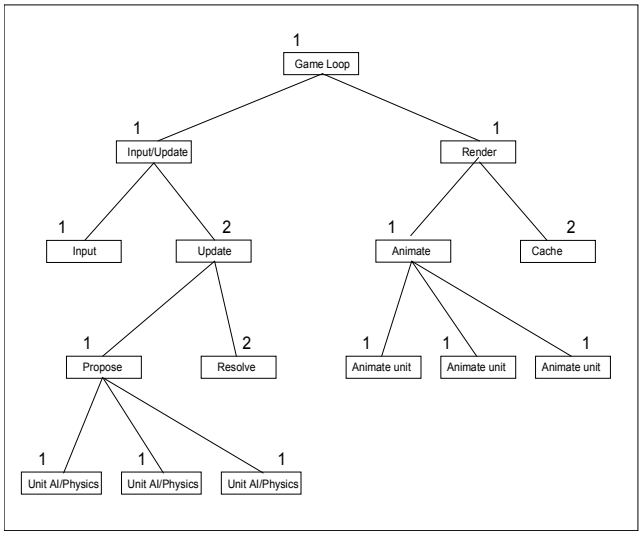

Figure 1. The task tree representing the basic game tasks executed during the game loop, and their scheduling order.

The task tree is a hierarchical scheduling representation of all the tasks that need to be executed during the game loop. Actual executable tasks are represented by the leaves of the tree. Each node or leaf of the tree is associated with a number representing its' scheduling order and nodes with equal scheduling orders are allowed to execute in parallel. Nodes with higher scheduling orders are not scheduled until all lower order tasks in the same parent node have completed execution. The task tree is processed to extract executable tasks into an execution queue which is serviced by a thread pool. As each task completes, the tree is reprocessed to extract further tasks which have become executable.

Figure 2 shows the speedups achieved using the task tree on three different machines using increasing numbers of threads. It can be seen that speedup is linear up to the number of processors of the machine.

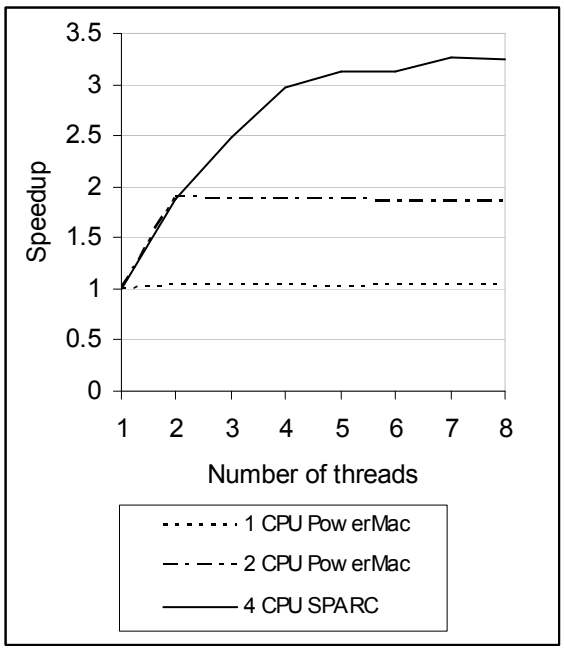

Figure 2. Speedup processing the task tree on three different CPU configurations.

The task tree approach is quite flexible, since the tasks that are assigned during each pass through the gameloop can be varied during construction of the tree. In addition, the approach is scalable to greater numbers of processors without having to modify the structure of the program. For example, the results of our test runs were all obtained using the same code base and varying only the number of threads used. The task tree also has 'semi-deterministic' processing characteristics, in that the order of task completion can be specified. All tasks in the tree will execute during a single gameloop. The approach is efficient, since threads in the threadpool are created only once and are reused for the duration of the game. Further the granularity of execution tasks can be varied to suit processor numbers. Finally, since there are no dependencies between concurrently executing tasks, no deadlocks can occur, no threads will block waiting on another task, processing of the tree is guaranteed to complete, and the processing resources of the machine will be fully utilized.

6. CONCLUSIONS

It would seem that a mixture of task and data parallel approaches is an optimum approach to exploiting parallelism in games. The problem in the design of multithreaded game engines is not so much in finding opportunities for parallelism, but in controlling

and scheduling execution of the many potentially concurrently executing operations.

This analysis of opportunities for parallelism in game engines indicates that the functions of a game engine can be separated into a small number of high level relatively independent tasks which operate in a producer-consumer pattern. Communication between these high level tasks can be organized as a set of information buffers. Within the render and update tasks there are sub tasks, some of which are strongly data parallel and some of which need to execute in sequence. The blend of opportunities for parallelism as well as sequential dependencies between tasks can be successfully encapsulated using the task tree concept. The task tree and threadpool approach we are investigating shows promise in terms of its efficiency, scalability, flexibility, and execution scheduling control.

Two areas of sequential processing are identified as key areas of further research. Identifying and organizing opportunities for parallelism within the process of detection and resolution of game entity interactions is a non-trivial task. Finding opportunities for parallelism in building the final render tree is another area which could repay further effort.

7. REFERENCES

[1] Andrews, J. *Threading Basics for Games*. http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/221160.htm 2005

[2] Breshears, C., Hoeflinger, J., Peterson, P., and Kerly, P. *Developing Platform Consistent Multithreaded Applications: Memory Management* http://www.intel.com/cd/ids/developer/asmo-na/eng/53797.htm 2003

[3] Coday, A., Magro, B., Breshears, C., Gabb, H., Kakulavarapu, P., Shah, S., and Tokinkere, V. *Developing Platform Consistent Multithreaded Applications: Application Threading* http://www.intel.com/cd/ids/developer/asmo-na/eng/53797.htm 2003

[4] ESA, *Essential Facts About the Computer and Video Game Industry.* http://www.theesa.com/files/2005EssentialFacts.pdf 2005

[5] Gabb, H., and Lake, A. *Threading 3D Game Engine Basics.* http://www.gamasutra.com/features/20051117/gabb_01.shtml 2005

[6] Haab, G., Gabb, H., Kakulavarapu, P., Shah, S., and Tokinkere, V. (2003) *Developing Platform Consistent Multithreaded Applications: Synchronization.* http://www.intel.com/cd/ids/developer/asmo-na/eng/53797.htm 2003

[7] IEAA, *Gameplay Australia 2005*. Center for New Media Research and Education, Bond University, W.A. 2005

[8] Llopis, N., *Introduction to Game Development, Chapter 3: Game Architecture.* Charles River Media, Massachusetts, 2005, 267-296

[9] Prensky, M. *True Believers: Digital Game-Based Learning in the Military*. http://www.learningcircuits.org/2001/feb2001/prensky.html 2001

[10] Sarmiento, S., *Real World Case Studies: Threading Games for High Performance on Intel Processors* http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/implementation/204081.htm 2005

[11] Sutter, H. *The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software*. Dr Dobb's Journal 30(3) March 2005