Game Engineering for a Multiprocessor Architecture

Abdennour El Rhalibi, Dave England and Steven Costa
School of Computing and Mathematical Sciences
Liverpool John Moores University
Byrom Street – L3 3AF Liverpool, UK.
a.elrhalibi@livjm.ac.uk , d.england@livjm.ac.uk , stevecostas@hotmail.com

ABSTRACT
This paper explores the idea that future game consoles and computers may no longer be single processor units, but instead symmetrical multiprocessor units. If this were to occur games would need to be programmed with concurrency in mind so that they could take advantage of the additional processing units. We explore past research and works in the field of parallel computing to find principles applicable to computer game programming. Concepts such as the Flynn's classification, task, task-dependency graphs, dependency analysis, and Bernstein's conditions to concurrency are applied to computer game programming to develop a new model for computer games that is meant to replace the standard sequential game loop.

Keywords
Concurrency; parallelism; threads; processes; game loop; task-dependency graphs; cyclic-task-dependency graphs; workers; tasks; queues; task manager; synchronisation mechanisms.

INTRODUCTION
Computer games have traditionally been programmed as a single loop (see Figure 1) which takes user input, performs necessary computations, updates the display, and continues again for the subsequent frame of animation. As computer games and the hardware they execute on increase in complexity, a new paradigm will emerge that will make single event-loop programs obsolete.
If computer architecture existed that featured multiple processors, it would behove the programmer to parallelise the game so that it can utilise every processor. The degree of parallelization would have to go beyond a simple concurrent I/O routine, a math function computed on a coprocessor, or a 3D scene rendered by a GPU.
With the introduction of simultaneous multithreading (SMT) in Intel's Pentium 4 processors there will soon be a widespread platform which can also take advantage of multi-threaded code. Yet the introduction of multi-processor computing may go beyond general computers. Rumours abound that both the PlayStation® 3 and the next XBox consoles will feature multiple processor architectures [1][4]. This new generation of consoles will challenge programmers to find new ways of decomposing their programs into parallelisable sections.
In an industry that continually pushes the limits of hardware; the single-loop game design must be discarded if symmetric multiprocessor machines (SMP) become the prevalent architecture for future game consoles.

Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play.

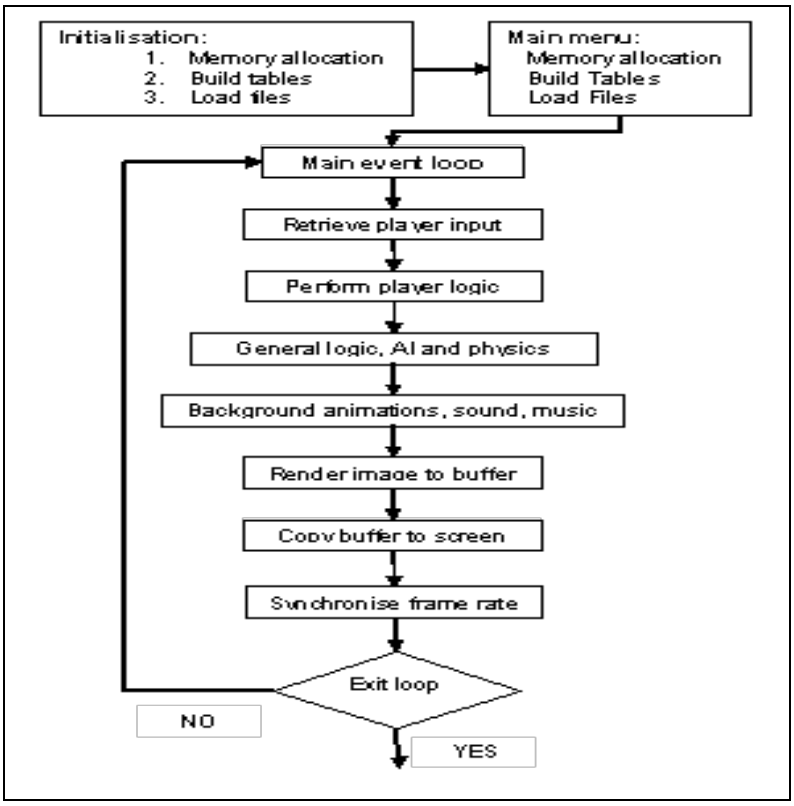


Figure 1: Traditional game loop

The structure of the paper is as follows: After the introduction, in section 2 we review Flynn's hardware classification, in section 3 we discuss parallel programming design, in section 4 we apply parallel programme design to our game example, in section 5 we evaluate the performance of the system, in section 6 we conclude.

MULTI-PROCESSORS ARCHITECTURE REVIEW

We review Flynn's classifications [1] [2][3] [5] [7] of hardware architectures for concurrent programming. The first of Flynn's classifications encompassed all computer systems with a single processor executing a single stream of instructions on a single stream of data [5]. He classified such systems as *single instruction stream - single data stream* (SISD). Readily available single-processor computers running a sequential program fall under this category.

Processors that execute the same program on different streams of data are termed *single instruction stream - multiple data stream* (SIMD) computers [5] [2].

The classification of *multiple instruction stream - multiple data stream* (MIMD) is assigned to systems where independent processors execute programs which work on different streams of data [5].

The final classification is termed *multiple instruction stream - single data stream* (MISD) [5].

Different hardware architectures could become potential target platforms Multi-processor computer or consoles. Architectures such as: *Shared Memory Multiprocessor Systems* [2], *Distributed Memory Multi-computers* [6], *Distributed Shared Memory* [2] or *Vector Computing* [3].

Shared memory multiprocessor systems and Vector processors are the most appropriate architectures for game implementation [9].

PARALLEL PROGRAMMING DESIGN

The main goal of a parallel program is to execute sets of instructions simultaneously in order to exploit the underlying hardware and achieve a performance gain above a sequentially designed program.

Task & Task Dependency Graphs

The units of computation a program is decomposed into are referred to as tasks [7]. An ideal parallel program will feature a set of tasks completely independent from one another and can be executed all at the same time. However it is often the case that some tasks rely on data produced by other tasks, and must wait for those tasks to complete their execution. A task-dependency graph is a data structure that can be used to represent such dependencies between tasks and their order of execution in relation to one another [7].

By analysing different configurations of a task-dependency graph for a given program, characteristics of its concurrent execution can be designed

Conditions for Parallelism

The following section presents two approaches which address the problem of identifying parallelism. *Dependency Analysis* looks to identify tasks which cannot be executed concurrently, while *Bernstein's Conditions* are a means of identifying sections of code (tasks) that can be executed in parallel.

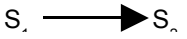
Dependency Analysis

Dependency Analysis is the process of identifying processes which can be executed simultaneously and those which must be executed in sequential order [2].

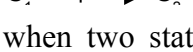
Different types of dependence between sections of code (tasks) exist which can hamper the opportunities for parallelism.

Data dependence, which is broken up into 5 sub-categories, is one of the main forms of dependence between program statements that hinders parallelism [6].

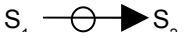
1. *Flow dependence* exists between two program statements when an execution path exists between S1 and S2, and at least one output from S1 serves as an input for S2.



2. *Antidependence* exists if a program statement S2 follows a statement S1 and the output from S2 affects the input for S1.



3. *Output dependence* occurs when two statements S1 and S2 output to the same memory location.



4. *I/O dependence* occurs when two statements access the same file.
5. Statement relationships that rely on data that is accessed by indirect addressing is categorised as *Unknown dependence*.

Control Dependence is another category of dependence that prevents parallelism between instructions [6].

Bernstein's Conditions

Bernstein's conditions [2] fall under the category of *Resource Dependence* [6], because they relate processes to the shared memory locations that they read from and write to.

They can be summarised as follows [2].

I_i is the set of memory locations read by process P_i

O_j is the set of memory locations altered by process P_j

Where ϕ is the empty set and $\cap$ is the intersection of two sets; Bernstein's equations are:

$$(1) \quad I_1 \cap O_2 = \phi$$
$$(2) \quad I_2 \cap O_1 = \phi$$
$$(3) \quad O_1 \cap O_1 = \phi$$

Once a sequential program is subdivided into smaller program components, it can be analysed using the aforementioned methods. By analysing the *data dependence*, *control dependence*, and *resource dependence*, opportunities for parallelism can be identified [2].

CONCURRENT DESIGN OF GAME

We have developed a *Concurrent Game Programming Framework* [9] for modelling games as cyclic-task-dependency graphs, and a scheduler to execute the tasks in a game in a scalable multiprocessor architecture. The framework features synchronisation primitives [8] (monitor, lock, semaphore, thread, thread worker, reader/writer lock, condition variable, etc…), thread creation objects, and classes for implementing games. The framework provides the programmer with solutions for synchronisation of shared resources, the creation of threads, and manages the execution of the game loop as a cyclic-task-dependency graph [9].

We explain below how a game can be designed to run on this framework.

General Game Description

The game created to experiment the *Concurrent Game Programming Framework* is a variation of 3D "capture the flag." The player controls a car and drives around a world featuring terrain and sky, as well as other cars which are controlled by AI system. See Figure 2.

Sequential Game Loop Analysis & Decomposition

Each loop-iteration of the game will involve the following instructions.

1. Poll the keyboard for user input.
2. Clear the display buffer and load the identity matrix into the rendering context.
3. Modify the scene matrix to reflect the user's camera view.
4. Update the user's frustum view object.
5. Render the sky background.
6. Render the terrain.
7. Update the position of each car in the game.
8. Organise each car in space partitioning structure then perform collision detection and response between each car.
9. Organise each flag in the space partitioning structure then perform collision detection between cars and flags to determine if a car has reached a flag.
10. Compute AI which searches the terrain for the closest flag to each.
11. Compute AI which guides each car to the flag.
12. Periodically check for those flags that have been captured and generate new ones.
13. Render the cars to the display buffer.
14. Render the flags to the display buffer.
15. Update the radar data.
16. Render the radar dial.
17. Update the top player list data, and check to see if the game is over.
18. Render the top player list.

19. Perform timing calculations to determine the elapsed time since the last flip of the display buffer; and flip the display buffer.

In a sequentially programmed game all of the tasks from 1 to 19 will be performed in sequence at each loop-iteration.

Figure 2: Screen shot of 3D CTF game used to evaluate the Concurrent Framework

Identifying Opportunities for Concurrency

With the game decomposed into sections, those sections must now be configured in such a way that some of them can be executed in parallel to one another.

An easier way to analyse the sequential game loop for opportunities for parallelism is to keep in mind that for Bernstein's conditions to be satisfied, two processes must be flow-independent, anti-independent, and output independent.

We will progress through each of the 19 tasks and determine their dependency relationships. Due to lack of space we describe the method applied only for tasks 1, 2 3, 4, and 5; for the full process see [9].

Designing with Dependency Relations

Task 1:

When polling the keyboard, the user input data must be taken and transferred to instructions which control car movement, camera controls, and registers if the user wants to exit the game.

Relation	Description
$T_1 \longrightarrow T_2$	User must be able to modify the camera view.
$T_1 \longrightarrow T_3$	User must be able to control his car movement.

Task 2:

Before any graphics-related instructions are computed the display buffer must be cleared, therefore all graphics tasks have flow dependence on task 2. It is important to note that although the task which flips the display buffer (task 19) follows task 2, task 2 has anti-dependence with task 19 because it cannot clear the buffer until it has been flipped.

Relation	Description
$T_2 \rightarrow T_3$	Camera matrix cannot be applied until matrix and buffer are cleared.
$T_2 \rightarrow T_5$	Sky cannot be drawn until matrix and buffer are cleared.
$T_2 \rightarrow T_6$	Terrain cannot be drawn until matrix and buffer are cleared.
$T_2 \rightarrow T_{10}$	Cars cannot be drawn until matrix and buffer are cleared.
$T_2 \rightarrow T_{11}$	Flags cannot be drawn until matrix and buffer are cleared.
$T_2 \rightarrow T_{16}$	Radar dial cannot be drawn until matrix and buffer are cleared.
$T_2 \rightarrow T_{18}$	Top player list cannot be drawn until matrix and buffer are cleared.
$T_2 \dashrightarrow T_{19}$	The Display buffer cannot be cleared until after the buffer has been flipped and displayed on screen.

Task 3:

In order to show the game from the correct camera angle, the modifications to the world matrix must be made before any objects are drawn. Therefore all objects that are rendered have a flow dependence on the modification of the scene matrix.

Relation	Description
$T_3 \rightarrow T_4$	Frustum object cannot be updated until after the camera has modified the world matrix.
$T_3 \rightarrow T_5$	Sky cannot be rendered until after the camera has modified the world matrix.
$T_3 \rightarrow T_6$	Terrain cannot be rendered until after the camera has modified the world matrix.
$T_3 \rightarrow T_{10}$	Cars cannot be rendered until after the camera has modified the world matrix.
$T_3 \rightarrow T_{11}$	Flags cannot be rendered until after the camera has modified the world matrix.

Task 4:

The frustum view object is critical for culling out geometry from being placed in the graphics pipeline if the user will not be seeing it. Therefore any graphics which rely on culling of objects before it is rendered must wait until the frustum view object is updated.

Relation	Description
$T_4 \rightarrow T_5$	Since the sky geometry will be culled it has flow dependences on task 4.
$T_4 \rightarrow T_6$	Since the terrain geometry will be culled it has flow dependences on task 4.
$T_4 \rightarrow T_{10}$	Since the car geometry will be culled it has flow dependences on task 4.
$T_4 \rightarrow T_{11}$	Since the flag geometry will be culled it has flow dependences on task 4.

Task 5:

Although the sky background has flow dependencies on other tasks, there are no tasks that actually depend on the rendering of the sky for their operation. Yet because the sky must render to the graphics card, it writes to a resource that is shared among all the other tasks that are involved in rendering (although task 2 and task 3 also share an output dependence relation this is not noted because it has already been shown that they share a flow dependence relation).

Relation	Description
$T_{\square} \ominus\!\!\rightarrow T_{\square}$	Two rendering tasks cannot take place at the same time.
$T_{\square} \ominus\!\!\rightarrow T_{\square}$	Two rendering tasks cannot take place at the same time.
$T_{\square} \ominus\!\!\rightarrow T_{\square}$	Two rendering tasks cannot take place at the same time.
$T_{\square} \ominus\!\!\rightarrow T_{\square}$	Two rendering tasks cannot take place at the same time.
$T_{\square} \ominus\!\!\rightarrow T_{\square}$	Two rendering tasks cannot take place at the same time.

Since all rendering tasks must write to the same location (the graphics card) they will all share output dependence relations with each other. For the sake of brevity, the relations will not be shown for each of the other rendering tasks and can simply be assumed.

The process is applied similarly to the others tasks. Figure 3 contains a completed cyclic dependency graph. It allows for all the nodes to be visited in a cyclical fashion much like in a typical game loop which must repeatedly perform the same tasks.

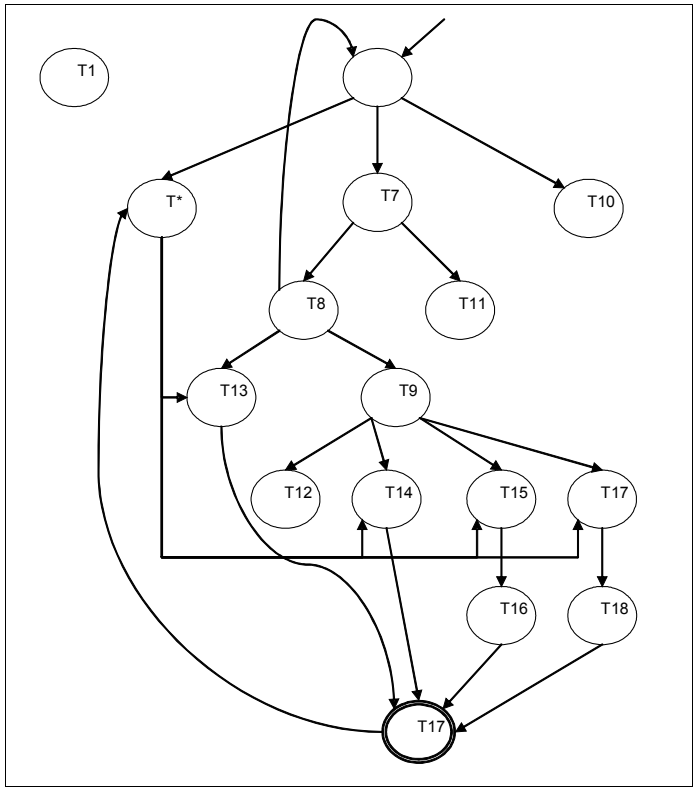
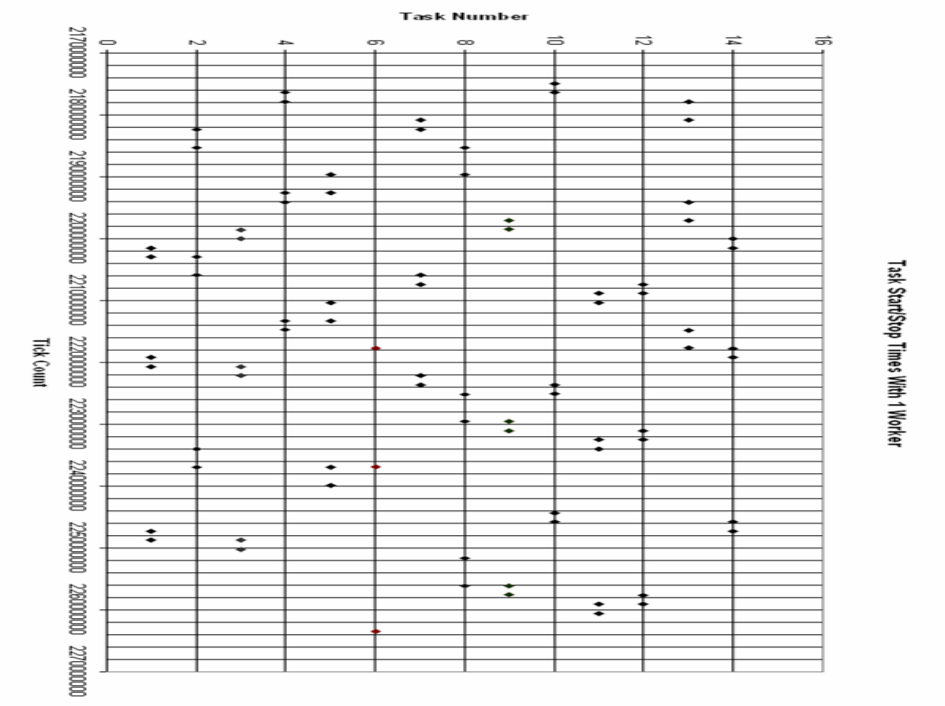


Figure 3: Game represented as a Cyclic Dependency Graph

Once the cyclic-task-dependency graph is obtained, it can be implemented within the *Concurrent Game Programming Framework* and take advantage of multiple processors in a shared-memory multiprocessor system. [9].

SYSTEMS EVALUATION

The performance results of the concurrently programmed game are assessed and compared with those of the traditional sequentially looped game, using simulation with The Simics simulation software on Windows® NT, and Hyper Threaded system.



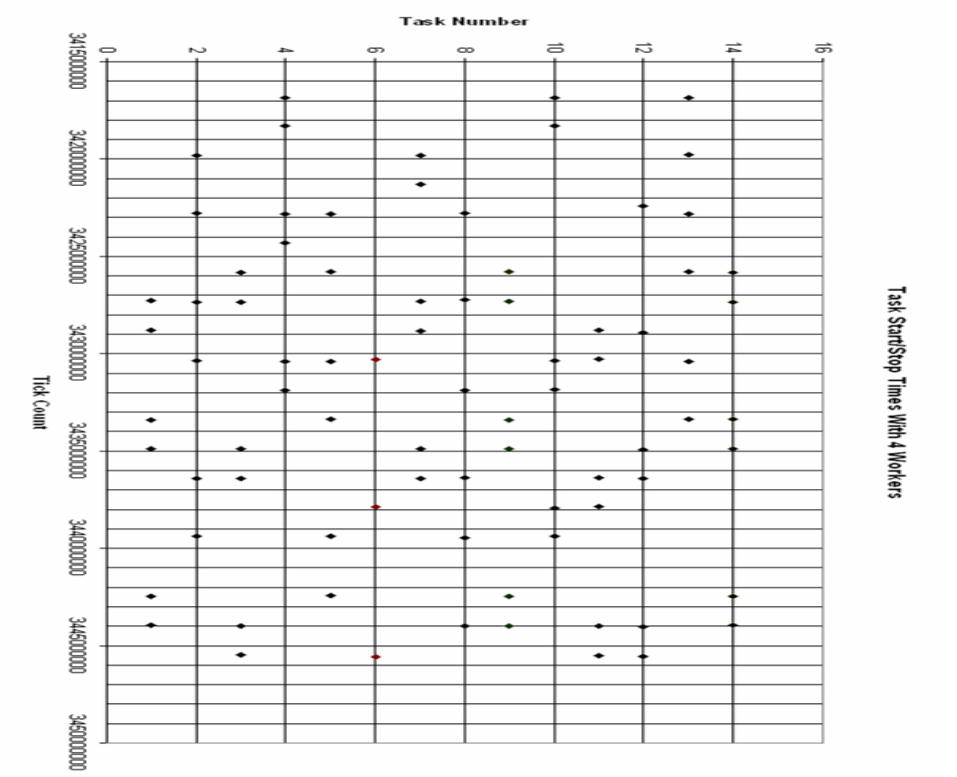


Figure 4: Evaluating the Concurrent Execution of Tasks

Test 1 – Evaluating the Concurrent Execution of Tasks

The first test (Figure 4) was used to evaluate the functionality of the *Concurrent Game Programming Framework*'s task manager working in conjunction with a game designed as a task-dependency graph being executed by 4 worker threads. By analysing the execution times of each task we could determine that the task dependency rules were being adhered to and the expected degree of concurrency was being realised.

By observing the two time graphs we can see that the framework was successful in parallelising tasks that were deemed appropriate for concurrent execution by the cyclic-task-dependency graph of the game.

Test 2 – Evaluating Performance on a Hyper-Threaded Machine

The second test (Figure 5) helped in the understanding of the limitations of using a task-dependency graph as a model for a concurrent game loop in a Hyper Threaded system.

To perform the test, 4 different game scenarios, with different maps, and number of cars, were chosen and executed 3 times for each of the configurations being tested. The different configurations that the game was made to run in were as follows.

1. Sequential loop (no framework)
2. 1 worker (game with framework)
3. 2 workers (game with framework)
4. 4 workers (game with framework)

All of the results for the different scenarios were averaged for each configuration. The system was tested on Intel Pentium 4 (Hyper Threaded) 2.80 GHz, 512 MB RAM, using Windows XP Pro, and featuring and NVIDIA GeForce FX 5200 (128 MB).

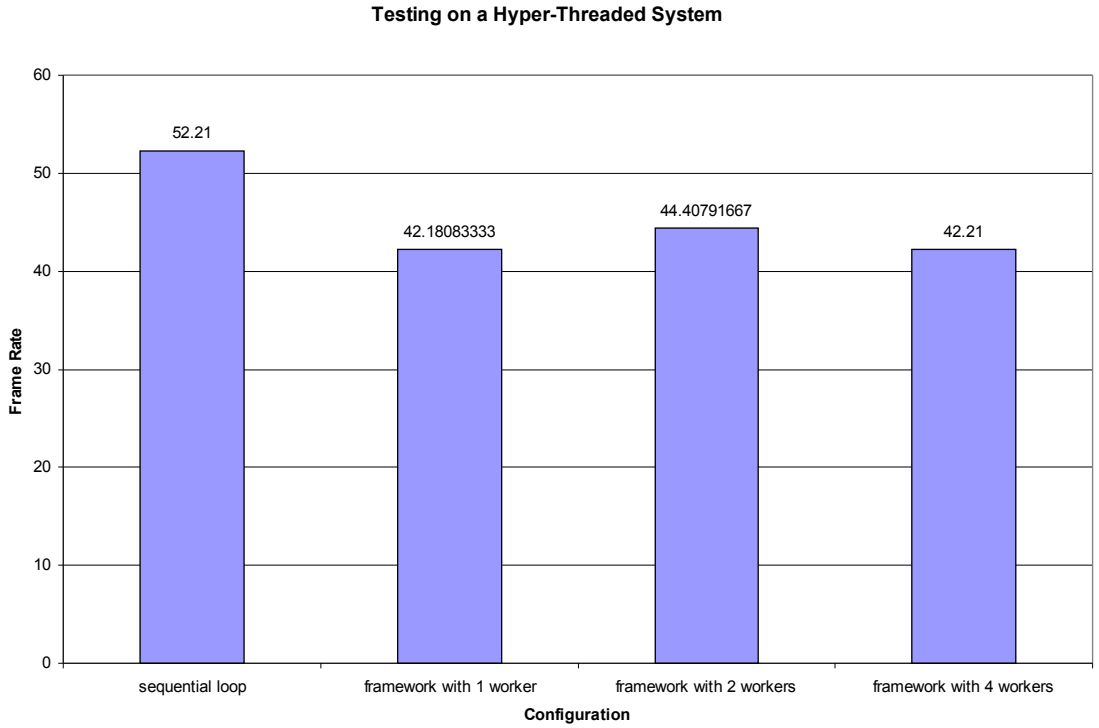


Figure 5: Frame rate results for a Hyper Threaded system

While a Hyper Threaded system provides 2 logical processors, the task dependency graph model requires 3 processors before a performance improvement can be expected over an equivalent sequential loop program. This explains the low performance.

Test 3 – Evaluating Performance on a Simulated Machine

The final test (Figure 6) helped to substantiate the claims of how a game redesigned as a cyclic-task-dependency graph using the *Concurrent Game Programming Framework* can utilise the available power of a shared memory multiprocessor system and show performance improvements over a sequentially looped game.

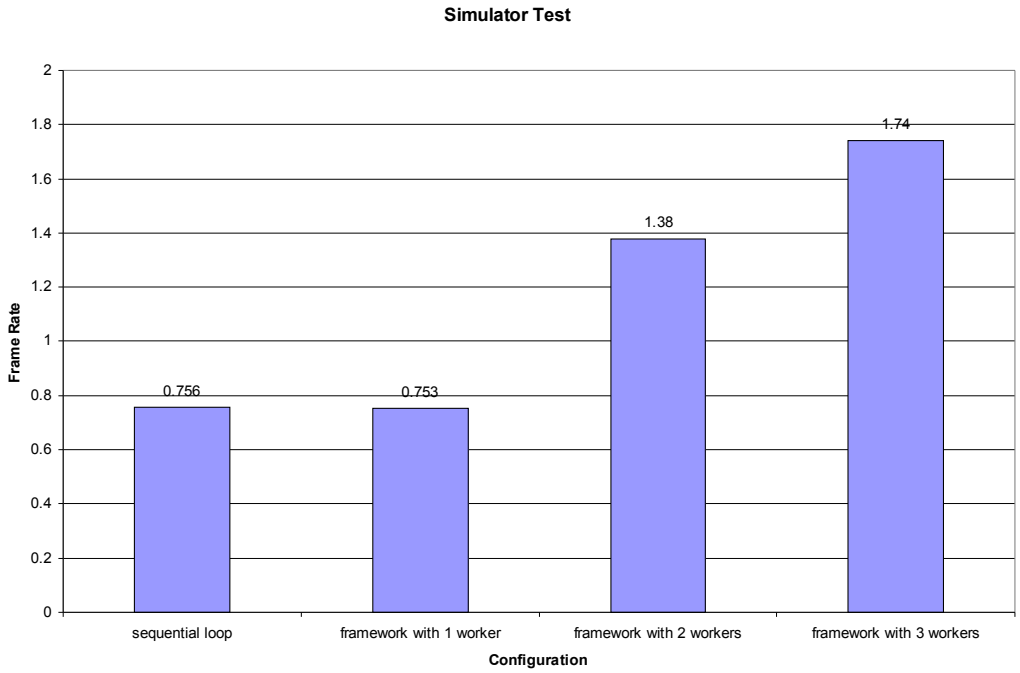


Figure 6 Frame rate results for a simulated 4 processor system

CONCLUSIONS

This paper was based on the belief that future game consoles and personal computers will feature multiple identical processors. Games programmed in the traditional sequential loop fashion cannot take advantage of shared memory multiprocessor hardware. This paper took the approach that the game loop should be re-structured and organised as fairly granular tasks in a cyclic-task-dependency graph using a *Concurrent Game Programming Framework* which features synchronisation primitives, thread creation objects, objects for modelling games as cyclic-task-dependency graphs, and a scheduler to execute the tasks in a game. A full 3D variant of capture the flags game, with car models, a terrain engine, car physics and car AI, is used to evaluate the framework. The Concurrent Game Programming framework developed allowed for a game to be structured and implemented as a cyclic-task-dependency graph. The tests performed to evaluate its performance capabilities showed that a game implemented in such a way and executed on a multiprocessor system could utilise the available processors and exhibit performance improvements. Furthermore, once the game is implemented it is scalable and can be executed on any number of processors by changing the number of worker threads. The test results collected have shown the framework to be useful in improving the performance of the system.

REFERENCES

1. Ragsdale, Susann, (1991), Parallel Programming, McGraw-Hill Inc., London, England.
2. Wilkinson, Barry, Michael Allen, (1999), "Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers", Prentice-Hall Inc., Upper Saddle River, New Jersey.
3. Fosdick, Lloyd D., Elizabeth R. Jessup, Carolyn J.C. Schauble, Gitta Domik, (1996), "An Introduction to High-Performance Scientific Computing", The MIT Press, London, England.
4. Sony Corporation, (2000), "EE User's Manual, Sony Computer Entertainment Europe", London, England.
5. M.J. Flynn, "Very High-Speed Computing Systems", Proceedings of the IEEE, vol. 54, December 1966.
6. Hwang, Kai, (1993), "Advanced Computer Architecture: Parallelism, Scalability, Programmability", McGraw Hill, London.
7. Grama, Ananth, Anshul Gupta, Georg Karypis, Vipin Kumar, (2003), "Introduction to Parallel Computing" Second Edition, Addison-Wesley, Harlow, England.

8. Hoare, C. A. R., (1974), "Monitors: An Operating System Structure Concept, *The Origin of Concurrent Programming from Semaphores to Remote Procedure Calls"*, Hansen, Per Brinch, pp 272 – 275, Springer, London.
9. Costa, S. (2004) "Game Engineering for a Multiprocessor architecture". MSc Dissertation Computer Games Technology. Liverpool John Moores University.