NEWCASTLE UNIVERSITY

Implementing Finite State Machines on the Graphics Processing Unit

Andrew Jones (110305163)

Master Computer Science, May 2014

Supervisor: Dr William Blewitt

Word Count XXXXX

**Abstract.**

Declaration

Acknowledgement

Contents

Introduction

Terminology

Introduction

Aims and Objectives

Research

Artificial Intelligence within Games

General Purpose computation on Graphics Processing Units

Finite State Machines using Shaders

Finite State Machines using CUDA

Developing a solution

Planning

Use of old code

Designing the core system

The Finite State Machine

Tools and technologies used

Developing the System

CPU

GPU

Optimisations

Testing

Before and after optimisations

Set test for CPU and GPU

Max test for CPU and GPU

Evaluation

Research and specification

Design

Implementation

Testing and results

Quality of tests

Data collected

Conclusion

Aims and objectives

What has be learnt

Further work

Summery

References

1. Introduction
   1. Terminology

*FSM*: Finite state machine

*GPU*: Graphics Processing Unit

*GPGPU*: General Purpose computation on Graphics Processing Units

*CUDA*: Compute Unified Device Architecture

*MMORPG*: Massively Online Role Playing Game

*HLSL*: High Level Shader Language

*XML*: Extensible Markup Language

*WoW*: World of Warcraft

* 1. Introduction

Artificial intelligence is a key element in the majority of modern video games and there has been many different AI systems researched and developed specifically for use within video games. These systems range from pathfinding to non-deterministic decision making and are very effective at what they do. Currently the vast majority of these systems are designed for use on a single CPU with very little parallelism in mind. Even though AI plays such an important role in these games it is slowly receiving less and less of the CPU’s processing time [1] to allow for more to be spent on graphics and physics processing. With these two limitations, AI is not advancing at the same rate within games as other aspects. Some games may require a large quantity of intelligent agents and with the current state of games AI this would not be possible.

If these systems could be made to be run in parallel, they would be able to be run on the GPU and it would make them much more scalable. The GPU has a parallel architecture and utilizing this would give an enormous performance increase, allowing for a large number of agents to be run at once. Over the years, there has be an ever growing interest in moving parts of video games onto the GPU but the main areas of research have been into implementing physics, complex mathematics and search algorithms on the GPU [2]. In comparison there has been very little research into implementing AI decision making onto the GPU however it has been attempted.

Massively Multiplayer Online Role Playing Games (MMORPG) have a massive number of agents within them, with many needing to be run at the same time. These agents generally run simple Finite State Machines (FSM) and rarely need communication between agents which makes them ideal candidates to try and implement onto the GPU. Shared memory is limited on the GPU and accessing shared memory can lead to less parallelism if used incorrectly so the lack of communication helps. Even though there is no communication there are still problems that may arise with just a simple implementation.

In this project, we will attempt to implement a modern day MMORPG agent on the GPU, see if it would give any performance improvements and what techniques could be used to give even further performance improvements. Firstly we will look at older and current attempts to implement FSMs on the GPU as well as what modern day MMORPGs AI systems involve. Then we look at the development of an AI system on the CPU being converted to a GPU system and what possible improvements can be made. Lastly we look at and compare the results of these different versions of the GPU implementation compared the CPU implementation and in which situations they are better or worse.

* 1. Aims and Objectives

The main aim of this project is to:

Develop a GPU implementation of a MMORPG AI system and compare how effective it is against a similar CPU implementation.

To achieve this aim, the following objectives were set:

* Research recent attempts at implementing finite state machines on the graphics processing unit
* Design a finite state machine for use within a MMORPG
* Implement a CPU version of the finite state machine
* Create a GPU implementation of the same finite state machine
* Develop potential GPU optimisations
* Asses CPU and GPU implementations over a range of agent quantities

1. Research

This section looks at an overview of modern AI within games as well as current programming on the GPU before specifically looking at previous attempts at implementing FSMs on the GPU.

* 1. Artificial Intelligence within Games

There are many different aspects to games and many different goals that games are created for however; the main aim of a game should be the give the player an enjoyable and fun experience. If there is a feature that takes a lot of resource but doesn’t increase this factor it will normally not be implemented or removed from the games to make room for elements that satisfy this factor more. This constraint is the main cause for the divide between research AI and game AI because game AI does not necessarily solve very complex tasks and needs to solve its problems in real time while taking as little resources as possible. AI still plays a key role in video games, however it has been placed on a lower priority for resources as realistic graphics and physics can usually give a bigger increase to player enjoyment in most cases. Although this is not the case for all games where very simple, or “stupid”, AI would ruin the player’s immersion. Using the GPU may allow for very complex AI systems to be implemented onto the GPU and still run in real time without hindering other areas of the game and in some cases could potentially allow completely new types of games to be created.

Another direction that GPGPU could take AI is instead of having more complex AI, is to have a massive number of simple AI all running in parallel. MMORPG need to have a lot of rather simple entities running at all time and sampling data from a high number of players and if there was a strong GPGPU implementation this could potentially vastly improve the number of agents they could use. One of the foundations of game AI is the FSM which is a power yet simple took for giving an agent in a game the illusion of artificially intelligence [3]. A classic example of a game that uses FSMs is Pac-man, in which each one of the ghosts has its own FSM to govern its plan on how to beat the player and where to move to accomplish this [4]. This simple yet efficient technique can be modified to emulate a variety of different agent types for use in a variety of games. As well as this, more complex AIs can be created using modified versions of this simple AI system such as hierarchical finite state machines or nondeterministic variants. These form a foundation for a high quantity of AI models in video games.

The AI for MMORPGs is normally some form of FSM that is ran on the server side thus it does not need to render anything at all which the GPU would normally do[5]. If the AI could be implemented onto the GPU this would allow many more agents per server or even potentially more servers to be hosted as well as freeing up CPU processing power for other aspects of the game. This potentially could allow more room for increasing the players experience within the game.

* 1. General Purpose computation on Graphics Processing Units

Manipulating the GPU for computations other than graphics rendering is not necessarily a brand new idea, there has been research into processing a range of computations over the past few years [2][6]. Originally developers had to exploit languages not made for GPGPU but gave them a basic platform to get started. These languages include Microsoft’s HLSL [7] and openGL shading language [8] and were originally made to use the GPU to processing graphical data. This allowed developers to explore the idea of harnessing the GPU’s parallel architecture using textures, vertex shaders and fragment shaders. Using fragment shader’s colour data, it was possible to send the agent’s data into the GPU and process it without the CPU needing to look at it [9]. With this method, over 2 million very simple agents were able to be created and ran simultaneously at 50 frames per second.

Recently however, there has been development of languages for General Purpose computation on Graphics Processing Units (GPGPU). These give the developers a strong platform to develop on that allows them direct access to the GPU. Currently, there are 3 main languages that are receiving the most attention, NVidia’s Computer Unified Device Architecture (CUDA) [10], the Khronos Group’s Open Computing Language (OpenCL) [11] and Microsoft’s DirectCompute [12]. CUDA will be the primary language that this project will look at and use however, the other languages will be looked at for comparison purposes.

CUDA was initially released in June 2007 and has been getting constant support from NVidia ever since, so in terms of computer science it is relatively new. In this period of time however, there has been a lot of research into developing on it as well as refining and optimizing it. Being developed by NVidia, CUDA is made to only be used with NVidia graphics card although, it is only newer cards that have compatibility with it and 2006 being the earliest generation. NVidia being one of the biggest graphics card suppliers; this doesn’t narrow down the audience completely though it does eliminate the console audience.

OpenCL was first made for use across heterogeneous platforms by the Khronos group although, it has since been picked up by Apple and now has been adopted by other big companies for use on their hardware. These companies include AMD, NVidia and Intel.

DirectCompute is Microsoft’s language for GPGPU which is used within their DirectX 11 framework and for compatibility works on DirectX 10 GPUs. This gives direct access to the GPU for development using the DirectX framework, which is very relevant for the games industry. It works on many different companies’ hardware although as it is part of Microsoft’s framework it requires windows Vista or newer operating systems to be used.

* 1. Finite State Machines using Shaders

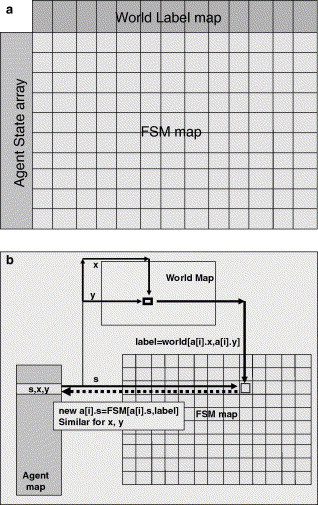
As finite state machines are a very common technique for agent AI, there have been a few different research topics into implementing them on the GPU. Even before GPGPU languages had been created people researched into attempting to create a FSM that worked on the GPU. Now languages exist, more and more researchers are attempting to implement a version of a FSM on the GPU.

As mentioned before, before GPGPU languages existed, developers had to exploit shaders to harness the power of the GPU. Rudomín [13], in 2005, developed finite state machine based agents using fragment shaders and was rather successful. Fragment shaders are traditionally used for per pixel colouring and for effects such as lighting. They allow the use of several textures as well as texture lookups, and this is the core part of what allowed a FSM to be created on one. Three maps are created from the textures, to store information about the world and the agents as well as the FSM.

Firstly a world map is created which is a normal map of the world. This map can contain any information about the world itself and there can be an arbitrary number of these maps defined by the designer. These maps may be collisions maps, height maps, normal or action maps.

The second map type is that of the agent space map. The size of this texture is the number of agents and the colour of the texture maps to the variables of the agent. For example the RGB values could map to the XYZ of the agent and the alpha could be the state the agent is in.

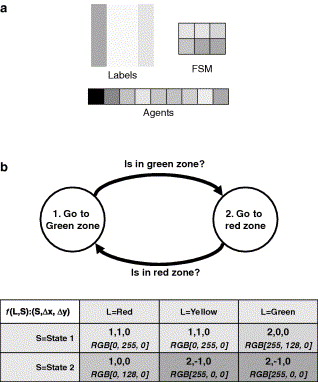
The last map is the FSM map. This map is a look up table where the agent’s state is maxed to the U value and the value received from the world map, at the agent’s XY position, to the V coordinate as shown in Fig1. Once these maps have been implemented, very simple finite state machines can be made onto them.



**Fig. 1.** How maps link to each other in the fragment shader via textures [13]

A rather simple finite state machine could be an agent patrolling between two locations on a map. For this the agent would have two states, one would be moving towards the first location and the other to move towards the second location. The FSM table would look at what state the agent is in and its position in the world. From these it could determine what direction the agent should move in. An example of this is shown in fig2.

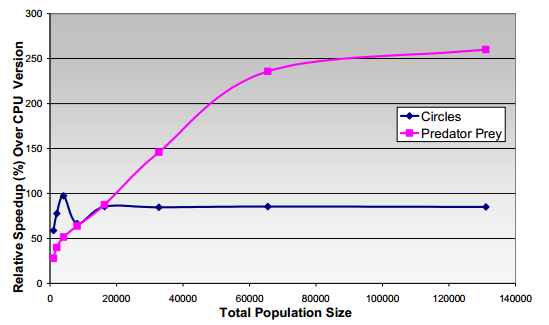
This solution works for simple FSMs and Rudomín also proceeds to create a hunter prey example using this system which uses more world maps and has 2 types of agents. The hunter can wander and chase the prey agents, showing that even more complex systems can be created this way.



**Fig. 2.** Simple patrol FSM and Maps [13]

* 1. Finite State Machines using CUDA

Since the development of GPGPU languages, there has been a lot of research into utilizing them for many different reasons. AI was no exception for this and as FSMs are a basis of games AI, people have attempted to reproduce them on the GPU. In 2009, Joselli and Clua managed to implement an entire game on the GPU using CUDA which used FSMs for their agents [14]. They came to the conclusion that even more complex AI should be able to be implemented on the GPU such as fuzzy logic or hierarchical state machines. Richmond, Coakley and Romano created an agent based model using Cuda and compared its speed with that of a CPU [15]. It compares two different models, Circles manages to stay around 90 times speed and the common predator prey model converges around 250 times faster than the CPU implementation shown in Fig3. This research shows how much better a GPU can potentially be over an average CPU implementation of the same type.



**Fig. 3.** Relative speed up of GPU Performance [15]

Later, Richmond also created a cellular level agent based simulation, once again using FLAME [16]. This simulation with FLAME allowed for a simple implementation of FSMs using XML model files. These model files specify core elements of the FSM system such as agent memory variables, states and transition functions that switch agents between states. There are also GPU schema extensions that specify variables such as the max population size, discrete agents, continues agents and communication between agents. With these model files, it removes the modeller away from having to fully needing to understand the architecture this allowing them to spend more time designing and developing the FSMs themselves. This is becoming a more and more important element of games development. Game designers should be able to add, remove and design elements of the game with needing to know as little coding because this speeds up game development as well as letting the designers have a more direct influence on the game.

Memory allocation is extremely important when dealing with GPGPU because there is not a large amount of shared data and the positions of the agents in memory can make a huge difference. Agents can be sorted into similar state based groups, this reduces discrepancies between agents, allowing specific look ups to be done at the same time and if some states have more branching than others it could have ruined GPU warping. If the agents are grouped together however, it is possible to estimate how long an agent in a group is going to take to be executed. At the end of the paper they also talk about how it could be improved further by increasing the detail of the cell models through the use of hierarchical modelling.

1. Developing a solution
   1. Planning

Before a solution could be developed, definitions of what the system must be able to achieve in order to satisfy its overall goals must be defined. These definitions are used for numerous reasons. Firstly they allow us to compare what is needed by the system to currently developed systems. They also make it easier to determine if the end project satisfies the goals it needed it to.

The system must allow:

* Numerous amounts of agents to be ran
* Player information to be stored and used by agents
* Finite state machines to be ran on the GPU
* Must keep a realistic update rate
* Maintain full functionality required by MMORPGs

Before the system could be designed, current systems needed to be looked at so that the system created in this project could mimic their behaviours and thus give an adequate comparison.

This project aims to develop a MMORPG agent FSM, so it makes sense to look at the most popular one on the market. In April 2015, 6.31% of all time spent gaming on the raptor platform was spent on World of Warcraft [17]. This is the most out of any MMORPG and second highest of all games on their platform. Their subscriber count in the first quarter of 2015 was 7.1 million and at peak, 12 million subscribers in 2010 [18]. Using the WoW statistics, gives a good basis to design and plan how many agents and players are typically in a MMORPG. A WoW realm (Server) at its peak could get to 5000 players when it had its max subscriber count, but more recently it has only been reaching just under 2500 at peak time on even the most populated servers [19].

The majority of agents in WoW don’t require communication between each other and just require player data. There are a few exceptions to this, such as an occasional agent will chase another, for example a wolf will chase rabbit or a group of agents may patrol together and be drawn into combat together. However, this is only the case for a small percentage of the agents, most just wander in their area or patrol between set locations. Jones & Bartlett [20] state that an average update rate for the AI system can vary depending on the game but an average update rate of 10 times a second is a decent rate.

These give a good benchmark for testing the developed system on. If the system can have that number of players and keep the realistic update rate, it can be counted as a success.

* 1. Use of Old Code

A base skeleton of a game engine, developed by Richard Davison and delivered to the students of the master’s course, was used as the basis of the system. The reason this engine was chosen was because it already had entities, physics and rendering implemented into it so it gave a good foundation for developing an AI system onto it. Parts of the engine where removed because the system did not require them. These parts were mainly physics based however, graphics rendering could have been removed also because a server will not need to render any graphical components as this will be done on the client side.

* 1. Designing the Core System

### The Finite State Machine

An exact version of WoWs current AI is not published so instead a Finite State Machine was designed with their one in mind, keeping the fundamentals in place. There are a wide range of agents in WoW however, most of them are very similar with only a few little differences between each other. The two most common agents are friendly NPCs that tend to stand in place and only attack players of the opposite faction and wandering hostile enemies that will attack any player if they come close.

The finite state machine was designed with the hostile AI in mind. There are 5 states in the designed FSM: Wander/Patrol, Look at Player, Chase Player, Attack Player and Leash. These states make up the fundamentals of most of the basic wow AI, the only thing that generally changes is the conditions for the state transitions or how and if they move around. Most, if not all, transitions are still player based although, limitations such as the player’s faction are added.

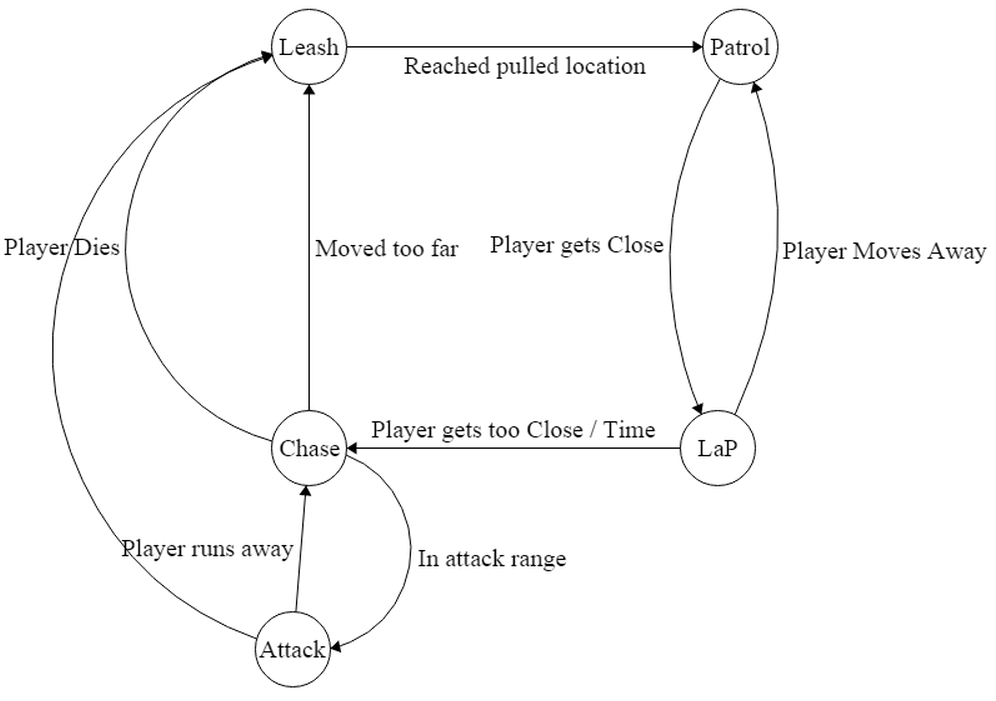
The first state the agent has and normally starts in is the Patrol state. In this state the agent moves between set points either predefined points for a patrol, or random points for wandering. There is only one state transition from this state is to Look at Player and that is when a hostile player gets within aggro range. Aggro range is calculated using the average aggro distance modified by the different between the agent and the players level.

Look at Player is a simple state where the agent just stops and stares at the player. If the player gets even closer to the agent or if the player stays in this range for too long, the agent will change into the Chase Player state. When it leaves this state, the position it was at needs to be stored for when the agent needs to leave combat.

Once the player has aggro’d the agent, the agent will chase the player until it gets into ability range with them. The same movement functionality will be used as the patrol state but the target point will be the players location.

All agents will have a basic attack, however a lot of agents will also have abilities that they should use. The abilities are cast in a priority order, with higher priority abilities being cast as soon as they can be, and the lowest priority being the basic attack. If the player dies, the agent enters the leash state but if the player runs further away it returns to the chase state.

The last state is the Leash state, this state is a reset state that brings the agent back to the location it was at before it was aggro’d. Once it reaches that location it returns to the patrol state.



**Fig 4**. Designed Finite State Machine

References

[1] Anderson, E. F. (2003). Playing Smart - Artificial Intelligence in Computer Games.

[2] Carneiro, T. (2011). A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU . 23rd International Symposium on Computer Architecture and High Performance Computing.

[3] Nareyek, A. (2004). AI in computer games. Queue, 1(10), 58.

[4] Thompson, T. (2008). An Evaluation of the Benefits of Look-Ahead in Pac-Man. IEEE Symposium on Computational Intelligence and Games.

[5] Douceur, J. R., Lorch, J. R., Uyeda, F., & Wood, R. C. (2007). Enhancing game-server AI with distributed client computation. Proceedings of NOSSDAV’07.

[6] Abshoff, M., Pernet, C., & Days, S. A. G. E. (2008). Efficient exact linear algebra over GPU. Presentation, August.

[7] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007, March). A Survey of general‐purpose computation on graphics hardware. In Computer graphics forum (Vol. 26, No. 1, pp. 80-113). Blackwell Publishing Ltd.

[8] Kessenich, J., Baldwin, D., & Rost, R. (2004). The opengl shading language. Language version, 1.

[9] D'Souza, R. M. (2007). SUGARSCAPE ON STEROIDS: SIMULATING OVER A MILLION AGENTS AT INTERACTIVE RATES. Proceedings of Agent2007 conference.

[10] Nvidia, C.U.D.A. (2007). Compute unified device architecture programming guide.

[11] Khronos OpenCL Working Group. (2008). The opencl specification. version, 1(29), 8.

[12] Thibieroz, N., & Cebenoyan, C. (2010). Directcompute performance on dx11 hardware. In Proceedings of the Game Developers Conference.

[13] Rudomín, I. (2005). Fragment Shaders for Agent Animation using Finite State Machines. Simulation Modelling Practice and Theory, 741-751.

[14] Joselli, M. (2009). GpuWars: Design and Implementation of a GPGPU. 09 VIII Brazilian Symposium on Digital Games and Entertain.

[15] Richmond, P. (2009). A High Performance Agent Based Modelling Framework. International Conference on Autonomous Agents and Multiagent Systems.

[16] Richmond, P. (2010). High performance cellular level agent-based simulation with FLAME for the GPU. Briefings in Bioinformatics, 334-347.

[17] <http://caas.raptr.com/most-played-games-april-2015-grand-theft-auto-v-muscles-in/>

[18] <http://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/>

[19] http://www.warcraftrealms.com/activity.php?serverid=671

[20] Buckland, M. (2005). Programming game AI by example. Jones & Bartlett Learning.