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

Agents and Players

Developing the System

Core System

CPU

GPU

Optimisations

Less Data

Broadphase

Optimisations for Parallelism

Modified Broadphase

Split Transitions and State

Sorted Data

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

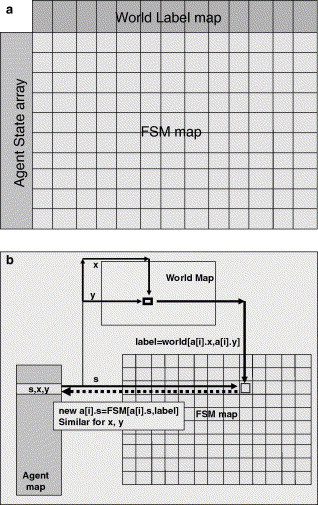
As finite state machines are a very common technique for agent AI, there have been a few different research topics into implementing them on the GPU. Even before GPGPU languages had been created people researched into attempting to create a FSM that worked on the GPU. Now languages exist, more and more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

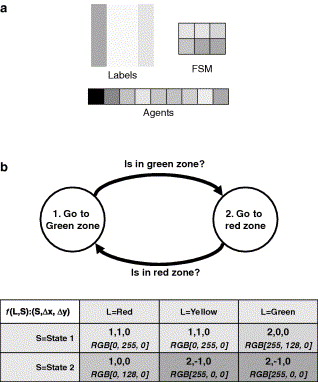
The last map is the FSM map. This map is a look up table where the agent’s state is maxed to the U value and the value received from the world map, at the agent’s XY position, to the V coordinate as shown in Fig1. Once these maps have been implemented, very simple finite state machines can be made onto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

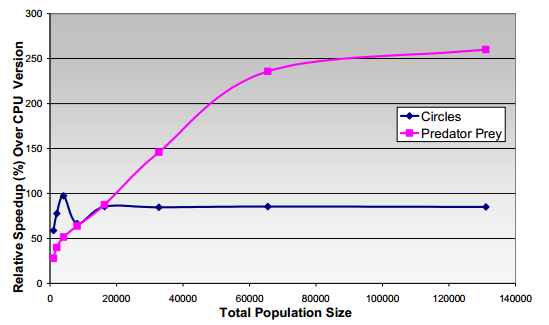
This solution works for simple FSMs and Rudomín also proceeds to create a hunter prey example using this system which uses more world maps and has 2 types of agents. The hunter can wander and chase the prey agents, showing that even more complex systems can be created this way.



**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. Languages, Tools and Use of Old Code

### Languages and Tools

C++ [21] was the chosen language for this project as it is the industry standard for games programming as well as having functionality with a range of GPGPU languages. C++ is also a lightweight language that doesn’t contain expensive unnecessary features which is what makes it ideal for games development but will also make it powerful for use on a server. CUDA [10] is the chosen GPGPU language for the development of this system as the tools that come with it are invaluable for testing purposes. Lastly OpenGL [8] was used for the graphical rendering, this was chosen because previous code had already been developed with it that allowed for more time to be spent on developing the key elements of the system.

Visual studio was chosen as the compiler for the development of this code because once again it is an industry standard as well as having simple support for CUDA. NVidia have developed a “Visual Profiler” [22] that allows intracet details about CUDA calls to be recorded and visualized. This allows the developer to see what aspects of their system are taking the longest, what needs improving and in which ways it can be improved. This software has been critical in the development of the system in this project.

### 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 WoW’s 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.

#### Patrol State*.*

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’s and the player’s level.

#### Look at Player State*.*

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.

#### Chase Player State*.*

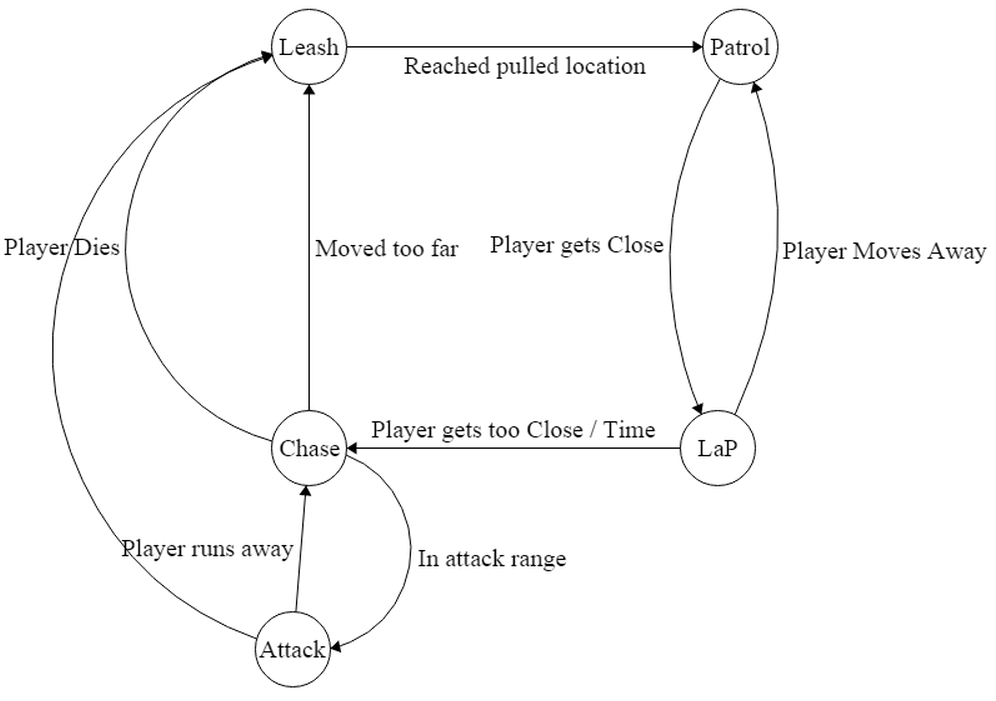
Once the player has aggro’d the agent, the agent will chase the player until it gets into ability range with them. The same movement functionality will be used as the patrol state but the target point will be the player’s location.

#### Attack State*.*

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. This allows the abilities to be stored in an array and the index of the array being the priority, 0 being the highest priority. If the player dies, the agent enters the leash state but if the player runs further away it returns to the chase state.

#### Leash 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

### Agents and Players

These agents are meant for use within a MMORPG so they require all stats, abilities and variables that a MMORPG will require from them. This doesn’t matter too much for the CPU implementation however, the GPU has much smaller allocated memory. As well as this, copying the data from the host to the device can create a rather large overhead. Each agent needs their XYZ coordinate, as well as a knowledge of what players to look at, the state it is currently in, the abilities it has, where it needs to patrol or wander to, its target player and even its level. With a large number of agents, these values start to add up if they need to be copied each frame.

The players could have the same problem as the agents however, not all the player’s information needs to be on the GPU for the agents to function correctly. The core data that the agents need to know about are the location of the player, their level and their health points. The quantity of data here is much less than the agents although it may still be worth looking into reducing the amount of data needing to be sent each frame

The way this data is stored is within structs of arrays. This means that instead of using an array or vector to store an agent class, it uses a single struct that stores an array for each variable the agent needs. This changes an agent from being a class, but instead being an index within the arrays. This allows for much higher parallelism and all the relevant data to be stored together. The arrays are stored in a struct instead of being separate because this allows for a single copy function call, which also has an overhead that becomes significant if used too often.

* 1. Developing the Game Engine and AI System

### Core System

A lot of the game engine has already be created however, for this project it needs to be modified and adapted for use with an AI system. The engine uses two threads, one for the graphical rendering and one for the physics. The graphical side was kept in the solution for testing purposes however in an industry solution this could also be removed so speed up the system further. As the system for this project does not require physics, they were removed and the thread instead runs the Artificial Intelligence Manager.

The AI Manager is the core class for managing the agents and the players. It stores all their relevant data and calls the relevant methods for the agents depending on the required implementation. It allows agents and players to be added from other classes to the system. It also updates where the graphical aspects of the agents and players are located although this can be removed for a server implementation that doesn’t require them to be rendered.

### CPU Implementation

The CPU implementation was the first to be developed so that a baseline could be made and so that all functionality could be created at the start for the GPU to attempt to copy. The CPU version was developed to try and closely as possible replicated that of a version a typical server may currently run. Each agent state had its own function, which contained its update for that state as well as the logic for its state transition. An array of function pointers pointed to these functions and the index of the array mapped to the state the agent was in. This saved a single switch statement being run as the agent just called the function it needed to at the time.

Even though this implementation was just for the CPU, the GPU was still kept in mind. The data structure for the agent and player storage stayed the same as if it was going to be a GPU implementation. The functions for the agents were also not class functions as the GPU does not support them. These modifications do not hinder the performance, and in some cases may improve it, however they do remove some easy functionality such as adding new agents or players however this is still possible and is not being tested.

With the high number of agents and players, as well as how spread out they are going to be, a broadphase was added so that each agent didn’t have to look through every single player when doing its checks. Uniform world space partitioning was implemented on the CPU, which told agents which partitions they were in and stored which players were also in the partition. This allows agents to just look at which players were in their partitions, vastly reducing the amount of checks they needed to do each frame. Uniform world space partitioning was chosen over octrees because of how separated the agents and players would be in the world. octrees could be used for very populated instances on the server, such as capital cities but because this is primarily world AI, it would have most likely not been worth the overhead. For the purpose of the partitioning, the agents and players were counted as spheres. The radius of the agents was determined by their maximum aggro range, this is because an agent from one partition could potentially be angered by a player in an adjacent partition. The players were not given a radius but rather just used their position. As well as this, a constraint was added that the partition size must be larger than the max aggro range so that an agent could only ever be in a maximum of eight partitions at once.

### Todo talk about kernels GPU Implementation

The first basic GPU implementation just converted all the CPU functions and functionality into GPU friendly functions. The GPU for example doesn’t use Vector3s so they had to be converted to float3s where used. Also function pointers were not used on the GPU so a function with a switch statement had to be added so that it knew which function had to be run.

As this was just meant to be a copy of the CPU version, all of the data was copied to the device and back to the host every frame which took a lot of time per frame. Each thread was associated with one agent (or index) and the number of agents was always kept as a multiple of 1024, as that was the maximum block size, giving it high block occupancy. Occupancy is the ratio of the numbers of active warps per multiprocessor to the maximum number of warps that can be active at once. This is not always a performance increase however, it is useful for when you want to gauge the latency hiding ability of the kernel [23]. From this basic implementation, it was already very parallelizable because the agents didn’t need to communicate with each other and a thread could just access its agent’s data without modifying any other agent’s data.

The broadphase was kept on the CPU with this implementation to give a fair comparison between the CPU and GPU implementation of the agent’s finite state machine although this started to prove to be one of the big bottle necks of this GPU implementation overall. As well as this, the partitioning data also had to be copied to the GPU every frame, adding an even larger overhead.

* 1. Optimizations

Through the course of development, several different optimizations were attempted to see in which cases the optimizations were worth using because some only become performance increasing at higher or lower levels of agents. Optimizations were attempted on almost all aspects of the system and not only for increased update rate but also, in some cases, allow for easier adaptation later.

### Todo Maybe add image here Less Data

The first optimization to be implemented was reducing the amount of data that needed to be sent over to the GPU every frame. There was a collection of data that only the GPU edited and a collection of data than the CPU edited as well as the GPU. With this in mind two structs were constructed. One struct was for data that only needed to be copied at the start of the application, which contained data such as the agents starting position, levels and any static variables. The second struct contained updated data that had to be sent each frame, this contained mostly variables from the broadphase but also any updating player data as that would not be calculated on the GPU. This reduced the amount of time wasted each thread copying data that did not need to be deleted in the first place.

### GPU Broadphase

One of the biggest bottle necks of increasing the agent count was calculating the broadphase on the CPU because of this, one attempt at optimizing the GPU implementation was to implement the broadphase on the GPU. This allowed for even less data to be copied onto the GPU, as well as some further parallelism. The first implementation of the broadphase on the GPU just used the logic for the default CPU implementation with some minor modifications. The agents and players broadphase was separated into different function calls, this allowed for different numbers of threads to be run for each kernel call.

For the player broadphase, a thread is run for each world partition and it loops through every player and compared them against itself. This means that there is no need for communication between threads, however it does mean there are only a low amount of threads being run and having time expensive for loops inside of them.

The agents were a little easier to convert while still keeping the same logic. Once again, after the players have had their broadphase ran, a thread is run per agent and if an agent is within a partition, that partition number is stored for that agent. This meant a lot more threads were ran than the player broadphase.

* 1. Optimizations for Parallelism

### Modified Broadphase

The initial conversion of the CPU broadphase to the GPU held vast improvements at higher agent counts, however the player broadphase still needed work. The maximum number of threads for the agent broadphase always scales with the number of agents but the player broadphase is static with the number of partitions. Parallelism is very important when programming on the GPU but so is scalability.

With this in mind, the player broadphase function was adapted so that a thread could be run for each partition-player check. This allowed for a significantly higher number of threads to be run in unison thus increasing the scalability, although this method does have a draw back. To accommodate this change and to stop thread racing, an atomic action had to be added for when the thread attempts to add the player’s index to the partitions list. Even with this reduction in parallelism, the update rate increase with this implementation was significant.

The idea of running a thread for each comparison was also applied to the agent broadphase. This this gave performance increases, even if not comparable to the player broadphase, however it has a rather large draw back. The maximum number of players the system runs is 5000 and the max number of partitions is around 100. Meaning that 500,000 threads are ran which worked with the hardware. The problem with applying this to the agents was that the agent count was reaching numbers over 1 million. Meaning that over 100 million threads needed to be ran for the agent broadphase and this was not possible with the hardware being use. This did not mean that it wasn’t worth doing as at lower agent counts, this was still a very viable improvement over the original implementation.

### Split Transitions and State

The next optimization was to the finite state machine itself. The state functions were split into two different functions per state. One function runs the logic for the state, for example where to move next or what ability to use. The second function was the logic that dictated if the agent should change state and to which state. The idea behind this was so that the functions could be called in parallel or on different updates. For example the state logic could be ran twice for each time the transition function was ran as its update rate is more important at higher update rates.

With this modification, a lot of different possibilities for optimization become available with next to zero overhead involved.

### Sorted Data

The last modification to the system wasn’t directly an optimization however, it potentially allows for many further optimizations to be made later. Richmond [24] states that it is potentially worthwhile to group agents via their state as this reduces divergence and ensure agents can be processed efficiently. As well as this if the agents are in sorted order in storage, it ensures that memory accesses from consecutive threads are linear and are therefore coalesced.

To achieve this, the CUDA Thrust library [25] was used to sort the data using the GPU. This allowed the agent’s state array to be sorted on the GPU and also to create an array filled with indexes that mapped to which states moved where and this was achieved by using the sort\_by\_key function. Using this index array, thrust was once again used to map all the other arrays, used by agents, to the correct sorted order. The second sort was achieved using the gather function in the thrust library. Each frame these functions were called to keep the data sorted however, once they were done the agent functions were all in sorted order and a count of how many agents were in each state was collect.

With this comes an overhead as well as possibilities. The overhead for sorting a rather large quantity of data is significant even when being done on the GPU. If you need the data to be sorted for an optimization this must be taken into account. If the optimization doesn’t improve the speed greater than the cost of sorting the data then it is not worth the time spent sorting.

In this project, one optimization with the sorted data was attempted. Divergence and branching can be a major problem when it comes to warping on the GPU. With the agents being in sorted order, agents of the same state can be run at the same time on their own kernel. Meaning the predicted end time is a lot closer than previous implementations and every agent could still have their own thread.

The problem with this was occupancy. There could be, at any point, an arbitrary number of agents in a state but to keep occupancy a set number of threads still needed to be run. This meant that “dummy” threads needed to be run, that either terminated shortly after starting or to still run through the function but have all modifications multiplied by zero. The reason it still ran through all the code was so that the predicted end time could still be closer to the average run time.

1. Testing

The main aim of this project was to develop a finite state machine for use within a MMORPG on the GPU. At the start of the project, key requirements were set to determine what the system must accomplish in order to be deemed a success. This section looks at these requirements as well as looking at how well each implementation and optimization worked. It will also go into detail on how the tests were conducted. There were 5 original criteria that the system must pass in order to be a success, these were:

* 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

Most of these criteria are simple and do not need testing, it either does it or it doesn’t accomplish it. There are two criteria that are worth testing however. These are numerous amounts of agents as well as realistic update rates. With these two requirements tests can be conducted and compared to see how each implementation improves over others in different situations.

* 1. Testing Environment

When conducting tests it is important to determine the testing environment so that all tests are fair and conclusions can be made. The tests were conducted on the same computer which had an NVidia GeForce GTX 780 Ti graphics card and an Intel i7-4470 CPU. The tests lasted 50 seconds, in this time the specific finite state machine was ran and the minimum, maximum and average updates per second were recorded. The number of players was kept at 5000 for all the tests to see how well the implementation runs in the worst case however, the number of players in a real implementation would be much lower than this.

Four quantities of agents were run for each implementation and each quantity was tested 3 times and the average was determined from this. The average update rate is the most important however, it is still crucial to see the minimum and maximum because player experience can be ruined if the minimum value is low and consistently being hit.

Fifty seconds was chosen as the testing period because this allowed the majority of agents to go through all the states of the state machine and have times in which the majority of agents were attacking a player as well as a time when the majority of agents were just in the patrolling state. Without adding the functionality of adding additional players, this is the optimal time frame for testing every state.

Nine different combinations of optimizations were tested to see which worked best at different quantities of agents. These implementations were:

* Basic CPU implementation
* Basic GPU implementation
* Copying less data optimization
* Basic GPU with original GPU broadphase
* Basic GPU with player broadphase optimization
* Basic GPU with player and agent broadphase optimization
* Split state and transition functions (CPU broadphase)
* Split state and transition functions (optimized player GPU broadphase)
* Sorted data optimization with optimized player GPU broadphase
  1. Test Results

Tables were formed for each test, which contains the maximum, average and minimum update rates. Graphs were also created from this data, one that shows the average update rates for each set of tests and one that shows the percentage of fps compared to the CPU implementation. Shown in fig5 and fig6. The full test results have been included in the appendix of the dissertation.

**Fig5**. Average updates per second

**Fig6.** Comparison to GPU implementation

* 1. Data Analysis

### Basic GPU verses CPU

The data collected allows for interesting analysis of the different implementations and optimizations. The overall goal was to create a GPU version of the CPU FSM and to determine if it was a performance increase and when it was worth doing. For a test to be classified as a success it needs to stay close to or above, preferably above, the ten updates per second mark on average. Comparing the basic CPU implementation to the basic GPU implementation, the CPU actually beats the GPU at 1024 agents and is generally close at 10,240 agents and has a much higher minimum update rate. However after this number of agents the basic GPU’s average update rate doesn’t decline as dramatically as the CPUs. The minimum for the GPU stays generally low compared to the average and this is probably due to needing to copy large amounts of data and the main bottle neck being the broadphase on the CPU.

Even with the base GPU implementation it is noticeable at high agent counts that it is superior to the CPU implementation as it keeps above 10 updates per second even above 100,000 agents and with the addition of the optimization that copies less data, the update rate raises even higher.

The real power of the GPU is shown once the broadphase is moved onto the GPU as well. Once the broadphase was being calculated on the GPU, close to 10 updates per second come be managed around one million agents. This made the GPU broadphase implementation of the FSM run roughly at around 14 times faster than the CPU implementation at this agent count. It can be seen that the GPU is faster in most cases, it’s just the 1024 agent tests that proved otherwise. Next we will look at, in detail, how the extra optimizations change these results further and delve into when they are better to use and when they are worse.

### Player and Agent Broadphase Optimizations

The first optimizations to look at are the changes to the agent and player broadphase. As can been seen by the test data, the changes to the player broadphase vastly improve the update rates at lower agent counts but eventually balances out with the original implementation because the time taken for players becomes insignificant compared to the agent broadphase and FSM.

When the optimization was also added to the agent broadphase, the initial update rate was slightly faster still, reaching 333 average updates per second. Although from the data it can be observed that the higher the agent count the slower this implementation gets because even though more threads are being run per agent in this test, they are limited by the fact they have to atomically modify a value. This means that threads need to wait for each other to finish modifying a value before they can modify it themselves which slightly removes the parallelism of the function.

If your system has a low quantity of agents or players these optimizations will vastly improve the update rate however, there is a cutoff point around 100,000 agents where the original broadphase implementation on the GPU starts becoming superior.

### Sorted Data

From the test data it can be seen that this implementation is slower than other implementations however, it gives the developer vastly improved data to work with. All the agents are in sorted order as well as the developer knowing exactly how many agents are in each state. Even at 100,000 agents it stays above 30 updates per second which is much higher than the original CPU implementation. In these tests an attempt at allowing more accurate warp time was implemented to see if that could overcome the overhead however, that has not worked.

Even though this optimization does not directly increase the average update rate, it gives room for many different attempts at optimizing further than that which has been done in this project. If optimizations can be created using this sorted data and extra information that overcome the overhead that comes with it then this is definitely an optimization still worth doing. Another reason this may be implemented is if the update is still higher than required but the sorted data allows for extra functionality in another aspect of the agent.

### Split Functions

Words here when I work out why it is actually faster to do it this way :D

1. Evaluation of the Project
   1. Research and Specification

The research undertaken in this project was invaluable to the design and implementation of the system. The research delved into previous attempts at accomplishing similar tasks and gave a clear understanding of the logic needed for developing on the GPU. The research also gave insight into the requirements needed by games that would potentially implement the system developed so that it could be designed around them.

The specification stayed the same through the entire development of the solution and this helped guide it and keep it on track. The target of the solution helped it be made bespoke for the game but also stay abstract enough that it could potentially be modified for alternative use. It gave guidance on the quantities of data that could potentially be put into the system as well as a foundation to design the finite state machine off of.

Even though the research was of incredible help, there can always be more research undertaken. More research could have looked at more modern techniques of specifically using CUDA and the thrust library to try and fully harnessing the power of the GPU. More research in these areas could potentially have influenced the design of key elements of solution leading to better and more interesting optimizations. However, even with this is mind, the project developed a solution to the problem at hand.

* 1. Design

The design phase of the project was rather short in comparison to a lot of other places. There are a few reason for this, the first being that of the project deadline. The project had to be finished in a certain time frame and this meant that some phases at to suffer. Although this was the case for design, specific design choices were made with this in mind. The use of old cold meant that the project’s implementation could focus entirely on the solution system rather than the engine it was working on. This meant that more optimizations could be made later in the project. Another design factor that sped up development was that of using the main concepts of a preexisting finite state machine. The finite state machine had to still be designed however, the core fundamentals were already set by trying to replicate the finite state machine from a MMORPG.

If the project was to be done again, more time should be spent on designing all the functionality that could and should work with the system. As well as this, each optimization could have been planned from the start. In this project, the basic CPU and GPU implementations were developed then from looking at profiling, optimizations were planned. With better understanding the GPU, optimizations could have been planned at the start of the project and potentially sped up the development process further.

As all of the features were not thought out at the start of the project, the current solution does not accommodate adding new players to the system, even though this would be an easy task to accomplish. If it was designed at the start, this easy feature could have been added to the system early on but the later it got into development the harder it became. The system could be modified so that when a new player was added, they could be added to a slot of a disconnected player. Then on the next update, the data stored on the GPU for players could be updated. This would be a solution with only a small overhead involved.

* 1. Implementation

Overall the implementation stage went rather well, however the time management could have been better in some areas. A lot of time was spent on some optimizations trying to get little performances increases, further than the basic optimization, which did not potentially matter in the grand scheme of the project. If time was managed better, more optimizations could have potentially been created. Many more optimizations could have been developed after the development of sorting the data to attempt to prove that sorting the data is worth the overhead in certain situations or potentially overall.

The shared memory of the GPU was not explored thoroughly enough with the optimizations developed in this project. Using the shared memory of the GPU usual yield more cache hits and has faster read time. The problem with the shared memory is that it is very limited on the GPU compared to the CPU however an optimization attempting to harness this could have been developed with more time or forethought.

Another problem that arose in the implementation state was that the CPU implementation was created first without the GPU version being fully in mind. A lot of code had to be modified so that it worked on the GPU as well as the CPU to allow for fair tests to be carried out. This was only a major problem because of the time frame of the project. If the project was to be conducted again with the same time frame, it would most likely be beneficial to develop the CPU system with the GPU in mind. This is so that time isn’t wasted rewriting code and more time can be spent developing optimizations.

The graphical side of the system, which came from the engine, could have potentially been removed to more realistically replicate what the server would have to do. The graphical system was very light weight so it should not have affected the results too much however, it does use the graphics card which is the primary target for the tests.

* 1. Testing and Results

The last step of the project was testing. Overall the testing phase gave results in the areas needed and allowed for in detail analysis of the finding although, there were some elements to the testing that would potentially be changed if the project was to be repeated.

### Quality of Tests

Each test was carried out three times to attempt to get an accurate result however, if the time frame wasn’t as short, more tests should have been carried out to further prove the readings were accurate. Three tests can potentially remove any anomalous results although there is still the chance two or more of the results were not accurate. Ten tests per implementation and agent count should have been carried out over a course of a day or two this is because background programs could have affected the speed at which the implementation ran, so the more tests at different times could have removed this factor.

Another factor that should have been tested was the solution on different graphics processing unit. All of the tests were achieved on the same computer, thus the same GPU. The solution was developed for a server for a popular game so it can be assumed that the server would have a high range GPU like the one used in these tests. Even though this may be the case, it would have been beneficial to tests the solution on a range of NVidia GPUs to see which one would be the cheapest option and still achieve the goal. Different GPUs could also potentially perform greater with specific optimizations.

### Data Collected

The main data that was collected for each test was the minimum, maximum and average update rate. These pieces of data allowed for the determination of if the specific implementation achieved the goal at hand. As well as giving a little further insight into its worst and best case. The data that was collected was enough that it could be determined in which cases a specific implementation is better while also showing the downsides. GPU memory usage could also have been recorded or even how long every single kernel call took however, the tests were to determine if the solution was a success and the data collected proves that.

The visual profiler gave a lot of useful information on the use of the GPU. This data was concerning, if the kernels have high branch divergence or warp time, high instruction replay overhead as well as high global instruction reply. This data alongside how long each kernel call took to execute was a major help in determining which optimizations were better and why. This data was used more as a detailed view of each test rather than an overall idea because if it runs faster overall it doesn’t necessarily matter if the kernels have higher branching. This data was left out of most formal testing because of this reason however it was used in some cases to show why one optimization was better than another.

1. Conclusion

Aim and objectives

What has been learnt?

What went well?

What could have been improved?

Further Work

Summary

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.

[21] Stroustrup, B. (1986). The C++ programming language. Pearson Education India.

[22] Visual Profiler, (2011). NVIDIA corporation.

[23] <http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/>

[24] Richmond, P., & Romano, D. (2011). Template-driven agent-based modeling and simulation with cu da. GPU Computing Gems Emerald Edition, 313.

[25] Hoberock, J., & Bell, N. Thrust C++ template library for CUDA, 2011.