BDI Agents for Game Development *

Arran Bartish
Royal Melbourne Institute of Technology
GPO Box 2476V,
Melbourne, Victoria, 3001,
Australia
abartish@bigpond.com

Charles Thevathayan
Royal Melbourne Institute of Technology
GPO Box 2476V,
Melbourne, Victoria, 3001,
Australia
charles@cs.rmit.edu.au

ABSTRACT

The study of game related topics has long been the subject of research and development, however game Artificial Intelligence (AI) predominantly uses classical AI simulation techniques. The AI community continues to develop concepts regarding various types of agents, genetic algorithms, and others, while older computational models such as state machines continue to be used as the back bone for game AI development. In this research, we have attempted to shed some light on how the choice of implementation may affect the important factors such as performance, and complexity as the games scales up. We have used performance measures, cognitive activities of designers and software metrics to arrive at our conclusions. Our findings show that agents and finite state machines have their relative merits. We have shown that, complexity measured as a function of the number of behaviours was linear for agents and quadratic for FSM. Though run-time performance is comparable for a small number of entities, it degrades higher for agents.

General Terms

JACK, Bomberman, JAVA, Agents, BDI, State Machines, FSM, Games

1. INTRODUCTION

The objective of this research was to find which model is more appropriate for game AI, and under which circumstances one model might be chosen over another. Due to the frustration of the industry and of users with what seems to be behavioural simplicity of game AI, we must define the advantages and disadvantages of both traditional state machine based and a BDI agent game AI. These choices will be determined through performance, iterative modification, and software engineering criteria. The reason being that each of these issues is of key importance to game development, a major software development undertaking.

Over the past years games have experienced many improvements, unfortunately many of these have not been in game AI. This has lead to problems where released games are spectacular in either 3D graphics, or in user interface, but Artificial Intelligence seems neglected. This is the part that makes a game fun and challenging to play. Despite much academic research being devoted to AI, the games industry continues to use rudimentary computer science technology such as state machines to create the desired behaviour for game entities. Recently the mould has begun to crack as a number of game titles are changing tack and focusing upon AI rather than graphics and interface. This could be a sign of things to come especially given the popularity and longevity these titles have enjoyed since release.

Given this seemingly changing environment in the games industry, exotic models for AI are being explored, but few ever get incorporated into games. This is partly due to the industry's general comfort with existing models, and the ever increasing urgency to release games in a shorter space of time. To our knowledge there has not been a formal comparison of current game AI technology such as state machines, and what is considered exotic forms of game AI like the Belief-Desire-Intention (BDI) [10, 11] agent architecture.

A formal definition of the pros and cons of both current game technology and BDI agents, will bring us one small step closer to providing a higher quality of entertainment. Games provide unique environments to test new and exotic technologies that otherwise could not be thoroughly tested in real world environments.

The new games appearing in the market are using more human like characters and are incorporating forms of cognitive learning. Attempts have been made to incorporate some of the newer technologies such as BDI in game development. However no thorough work has been done to analyse the merits of traditional and agent methods.

2. BACKGROUND OF THE PROBLEM

This research required the development of a game to use as a test case. In addition to this two models were implemented to simulate the intelligent behaviour. One model was an agent, the other a state machine.

2.1 JACK

JACK is an agent programming language developed by an Australian based company called Agent Oriented Systems.

It is an extension to the JAVA API, which allows the use of several other base classes. These base classes include Plan, Event, Database as well as some others. JACK allows the simple and easy definition of simple intelligent agents and BDI agents, with very little trouble for the programmer. JACK also allows the programmer to use the normal JAVA API, a huge advantage considering the growing popularity of the language.

JACK is the base we used to create our Bomberman agent. The reason for this was two fold. The first and most obvious reason is the availability of JACK and the support from Agent Oriented Systems. The second being its relationship with JAVA to allow quick platform independent development. In addition to this, we could develop an agent model and state machine model that were both JAVA based. It would not be a justifiable comparison if the two models were based on separate languages. Using JACK allowed us to use inheritance and other object oriented techniques that gave us the flexibility to swap one model for the other as required.

In order for us to do a comparison on state machines and agents, we need to describe how a JACK agent chooses to execute plans. JACK seems to have taken inspiration from event driven programming, as JACK agent plans are dependent upon event-like goals. Our JACK agent uses rule-based evaluation (similar to the state machine) to decide which event should be fired. When an event is fired the relevant plans are initiated and can be executed. So in this way events are much like goals in that when an event is fired, plans are executed to achieve the goal represented by the fired event. JACK allows a lot of versatility in this matter, and plans can evaluate their own suitability before they execute. Many JACK features such as plan self-evaluation and teamwork, were left out to allow for a *fairer* comparison between state machine and agents.

Unfortunately JACK also has a number of problems. Some of these problems were cited by John Thangarajah [14]. He highlights a problem with JACK in its inability to represent goals in a traditional sense, a fundamental concept for BDI agents. He continues to describe his proposed addition to JACK to allow this feature. It must be noted that while this addition could be made in future research, we have used unmodified JACK as this would suffice for the purposes of this research. The version we used was JACK 3.2 released in the middle of 2001. For more information on JACK please see user guides [6, 5]. For more information on JACK and teamwork see [7].

2.2 Games

Games programming involves many core areas of computer science. Increasing the computational requirements of games pose many problems. Some of the problems being tackled range from graphics clusters, algorithms for polygon limiting, artificial intelligence, and NP complete problems like "travelling salesman". This makes the game industry a fine test bed for innovative ideas and concepts at the leading edge of current technology. There has always been constraints forced on game AI, brought about through the need for graphics and IO processing, however these are becoming less of a problem as hardware for these purposes improves.

2.2.1 Finite State Machines

State machines have long been the most popular implementation of game entity behaviour [16]. State machines are easy to understand, and are a concept most computer science graduates have been exposed to. Students commonly use state machines for the parsing of grammars. What most of these students don't understand is that a state machine can be a very powerful implementation to simulate intelligent behaviour in games. They have been the traditional implementation for game AI for many years, and it appears they may be for many more. The basic characteristics of a state machine are:

- That it has a fixed amount of memory.
- A state machine is driven by input.
- It undertakes transitions between states.
- It produces simple output.

The concept of a finite state machine has long been developed and improved over the past decades to include variations such as fuzzy state machines and fuzzy logic. However these implementations will not be the focus of this research, and a traditional finite state machine will be used during the experiments. For an introduction to finite state automata or if the reader desires more information about the topic of computation, we refer you to [4] for an in depth introduction and description of finite state automata. State machines can be used to process complex and dynamic game environments. However they are prone to software engineering issues [15, 3], such as duplicate state decision points, that threaten to break even well designed state machines.

State machines are best described as deterministic by nature, and implementing them can lead to software engineering problems as mentioned above. There are three prominent ways to implement a state machine [3], and the choice as to which implementation was to be compared in this research was of utmost importance. The first is using a block of if, or switch statements which are executed on some condition or state. This method could easily lead to duplicate state decision points, a common flaw in this method. A duplicate state decision point can occur through poorly documented code, and results in conflicting transition conditions. In addition they blur the concept of state and transition where a transition or condition statement might contain the implementation of the state. Another implementation is a decision table, indeed a number of games have used this implementation. It is fast, however it restricts versatility and this is far from satisfactory given the dynamic environment of games. The third choice (and the one used for this comparison) is an object state machine, where a state, its implementation and transitions are encapsulated by an object. State transitions in the object state machine are represented by an engine that keeps track of the current state. This model has a slightly higher run-time cost, but it's conceptual advantages out weigh this cost.

2.2.2 Current Game AI Technology and Trends

Steven Woodcock sites at the 2001 Game Developers Conference that the majority of games being released still use a

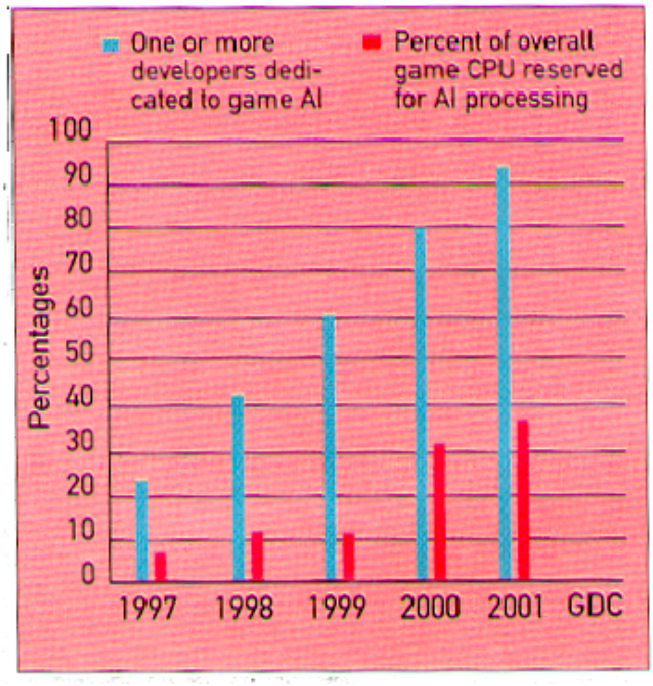

Figure 1: Current Industry trends with regard to AI resource allocation (Chart extracted from [16]).

state machine implementation [16]. This is interesting however, as the industry as a whole seems to be devoting more resources than ever before to game AI. In the same article, Steven Woodcock describes how the number of programmers per project devoted to AI has grown from 1 just 5-6 years ago, to a dedicated team of developers per project (see figure 1). For convenience figure 1 has been provided. He also discusses how greater amounts of CPU time are also being allocated to AI processing. This could be partly related to the fact that a lot of graphics processing that used to be undertaken by the CPU, is now being delegated to dedicated graphics accelerators. This would have the effect of leaving the CPU available for other tasks. Most developers feel that AI is growing in importance and will have a greater effect on games of the future [8].

State machines have a number of inherent problems. One such problem is referred to as duplicate state decision points as cited by [15, 3]. This is a software engineering issue that can be quite troublesome and could potentially break any State Machine. This problem usually occurs when code is not properly commented and documented. The result being states could be implemented that have conflicting or identical state decision points, or conflicting rules. Obviously this could cause a detrimental effect if your in-game tank suddenly finds itself in an incorrect state due to one of these conflicting rules. The second problem is more of a problem inherent in all state machines. Because state machines are deterministic by definition, the behaviour produced by a state machine is also deterministic. This leads to behaviour that is predictable. To avoid predictability, most developers are either using fuzzy state machines or fuzzy logic, even so these are still based on a model state machine model. Indeed most of the AI scripting that is being implemented for most games gets abstracted into some form of state machine. While more exotic models for game AI are being looked into, developers with ever decreasing schedules are finding that the time required researching these other models are beyond what they can afford. So in the tradition of commercial industry, many new and exciting techniques are being put into the too hard bin as developers resort to the tried and tested techniques such as the finite state machine.

As the average bandwidth a user has available for communication increases with connections to the internet such as ASDL and cable, games are starting to show a shift away from single user environments. Instead users are finding that games can be more challenging and generally be more fun due to multi-user aspects of the games. These games particularly are aimed at the multi-player environment and yet we find that AI could be just as important even though AI might not be required. Often users logon to a game server to find it empty just to leave and find another server. Another situation that might arise is where users find themselves in unbalanced teams, where the team of greater numbers dominates the game. What would be the result if believable BDI agents could be introduced to add some numbers to a realm or even out mismatched teams dynamically during game play and of course be removed when no longer needed? Such features would improve the experience of online gaming in general. The team aspect is of utmost importance when talking about games dependant on teamwork. In most cases AI is not required, however there are those times like those mentioned above, when plausible AI opponents would be of great value to the game. It must be noted that AI opponents can already be included most games, however these opponents are either ordinary in skill, or not plausible as a human player, ie too hard, or too easy.

2.3 A State Transition in Thinking

Recently there has been a real break from the established mould of game AI by the game Black & White by Lionhead Studios [8, 2]. The use of the BDI architecture as well as a number of cognitive learning skills show just how appropriate agent concepts are to these highly dynamic virtual environments. Included in Black & White are other agent concepts such as agent emotions, where game characters can become depressed and exhibit human reactions to these emotions such as binge eating. Obviously this adds whole new dimensions to the game experience. It could be that this is the start of things to come in game AI, citing games such as Black & White, The Sims, and others with sophisticated user interaction as proof of concept. Just as graphics have been a draw card for most games in the past, sophisticated AI could be a draw card for games being produced in the future. It is hoped that by producing games with greater intelligence, game characters will challenge the user and keep their attention for a greater period of time than they do currently. This change in focus shows that the game industry is beginning to mature from the 3 friends in a garage to an industry that is relying heavily on research and creative thinking coming from the academic world.

The way in which Game development is changing also reflects the growing focus of AI in games as figure 1 shows. There is a trend that most development teams are allocat-

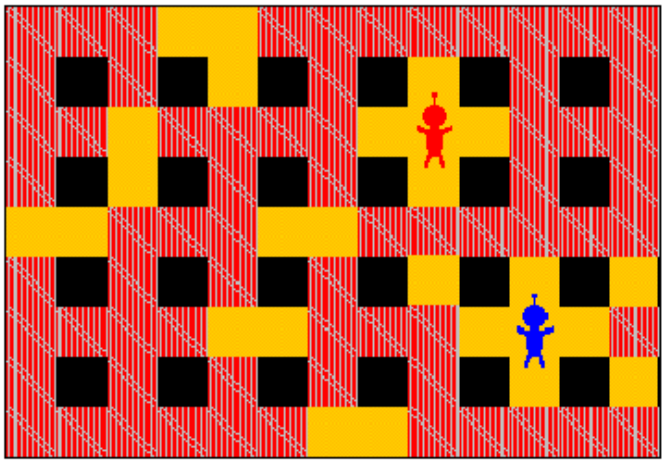

Figure 2: Blue and Red Bombermen at initial Starting locations.

ing at least one or more personnel specifically to AI which is in contrast to only a few years ago, where game AI was hacked together in the last month or so of the project by anyone who had time. In addition to this, as mentioned before, AI is being allocated more CPU time than it ever has been previously. This, coupled with the increased allocation of human resources to AI development, makes it clear that developers are giving AI a much higher priority than it has enjoyed in the past. We can therefore infer that we will see more of the sophisticated AI prevalent in Black & White in up future games.

2.4 Simulation Vs Game

There is a difference between a simulation and a game. However sometimes that line blurs, making the distinction difficult to identify. We find that simulations are often based on real world environments and physics. A game however, will embellish those truths in the interest of user entertainment. When someone decides to create a simulation, the object is to model something as close as possible as it is in the real world. The line that distinguishes simulations and games blurs during the design of a game when designers decide to make a game both enjoyable to the user and keep behaviour in the game realistic. Past research has seen agents used in battle simulations [12, 13] for teamwork, and in games such as Pengi [1]. Even with research such as this, it seems that very little investigation has been done on direct comparisons between developing agent architectures such as BDI cognitive agents and the classic game AI implementations of state machines.

2.5 Atomic Bomberman

Atomic Bomberman was a game produced by Interplay in the mid 1990's. Atomic Bomberman is a recreation of a much older game that has long been popular and was first released around the same time as Pac Man. It is a 2D game with an obvious state machine implementation, which often leads to predictable and simple behaviour. Figure 2 shows a possible start of game situation for a blue and red Bomberman. The objective of the game is to remove crates and create a path toward the enemy. Following this, they must try and trap their enemy and blow them up. A problem that faces the Bombermen is to move toward the enemy, without blowing themselves up. When contact has been made they must attempt not to get trapped themselves while endeavouring to trap their opponent. Bomberman is a dynamic environment that is changing constantly. This fact makes Bomberman a perfect candidate to test agents. It must be noted that this research is not about Bomberman, but a direct comparison of the two models. Hence the game is subject to change in order to produce more accurate experimental results. Indeed changing the game by adding new game entities, and analysing how it effects the software development process, is an important part of this research.

3. CODE AND DESIGN COMPLEXITY

Game development typically follows a prototyping approach where new behaviours are added until a playable game that can sustain the users interest is created. Thus it is important that any methodology used lends itself to such an approach. At the design stage this may involve adding new behaviours and changing the interactions among the entities. At the programming level this involves adding more logical paths which naturally adds to the complexity.

3.1 Design complexity

To measure the design effort involved objectively, we have devised a control experiment where a design diagram had to be changed to reflect a new game entity. Six students from RMIT volunteered to take part is this experiments. All of them have encountered state-machines in their course of study and had some understanding of agent technology. All participants received the same design problems - enhancing the Bombermen with an additional terminator entity. They were also provided with short description of Bomberman details of the design specifications. Figures 3 and 4 show the original state machine and agent definitions.

A simple scoring system was created to grade the quality of designs into 3 categories, good, workable and wrong. The time participants took to complete the design was noted down. Based on our criteria all the agent extensions were correctly identified and had little or no variation among them. Of the state-machine extensions, one was graded good, three were workable and two were found wrong. Most commented that they found the state machine version required a lot more analysis while the agent version was very similar to human reasoning. The time taken by the various participants are listed in table 1. Note, to ensure fairness we made three of the participants complete the agent design first and the others vice-versa. On average, the design changes to state machine version was 4.5 times faster.

	State Machine First			Agent First		
	P1	P2	P3	P4	P5	P6
State Machine	7	8	4	4	2	2
Agent	1.5	1.5	1	1	1	0.5
Ratio	4.7:1	5.3:1	4:1	4:1	2:1	4:1

Table 1: The time taken to make modifications in minutes

The designs in figures 5 and 6 were the most representative

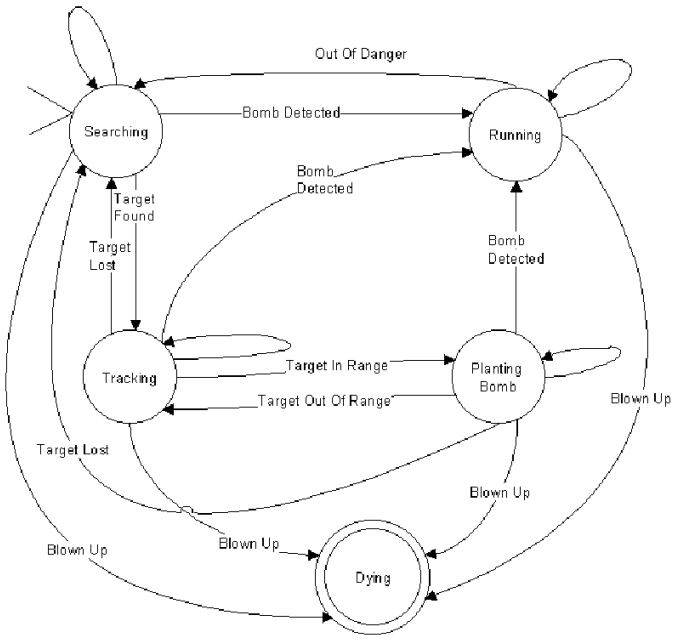

Figure 3: Initial State Machine definition for Bomberman before modification

Agent Goals	Corresponding JACK events	Agent Plans To Execute
Escape Bomb	Bomb Detected	Escape Bomb
Move To Target	Target Out Of Range	Track Target
Find New Target	Target Not Found	Find Target
Blow Up Target	Target In Range	Place Bomb

Figure 4: A Simple agent definition that novice Agent designers can quickly understand.

of the extended versions.

Note that the state machine version has undergone extensive changes reflecting the high coupling of this design. The design change to agent version, merely involves an additional goal with a corresponding event and plan.

3.2 Code Complexity

We then attempted turning our representative design into equivalent code measuring the additional changes involved. Table 2 shows the changes involved.

Model	Total Classes Modified	Total Changes
Agent Module	3	7
State Machine Module	6	10

Table 2: A summary of all modifications required to add a simple change to behaviour. (See appendix for more detail

To measure the programming complexity, objectively, we have used McCabe's Cyclomatic Complexity metric. Introduced by Thomas McCabe in 1976, it measures the number of linearly independent paths through a program or module. It is a widely used metric for measuring soundness and confidence of a program. As the McCabe's cyclomatic complexity values below shows, higher the complexity of a design, the harder it is to maintain, and higher the risk of introducing bugs.

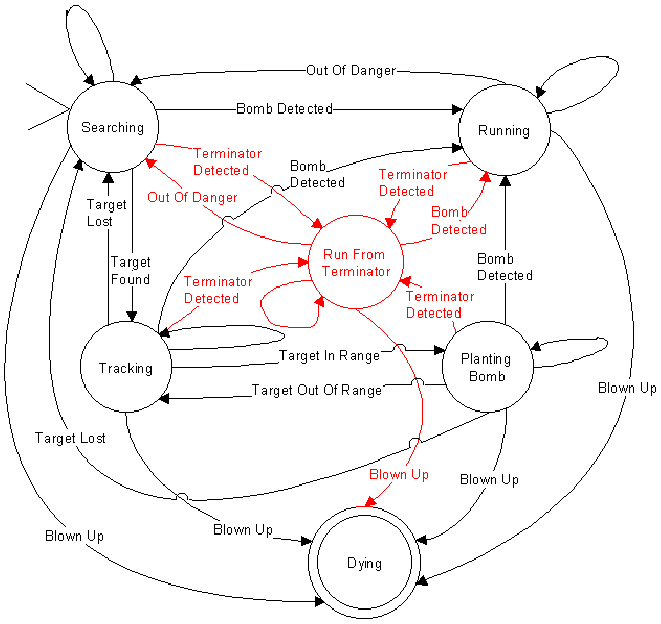

Figure 5: State Machine definition for Bomberman after modification (Changes shown in red)

Agent Goals	Corresponding JACK events	Agent Plans To Execute
Escape Bomb	Bomb Detected	Escape Bomb
Move To Target	Target Out Of Range	Track Target
Find New Target	Target Not Found	Find Target
Blow Up Target	Target In Range	Place Bomb
Escape Terminator	*Terminator Detected*	*Run From Terminator*

Figure 6: A common design modification produced by experimentation

We then applied the McCabe's cyclomatic complexity equation for both models before and after incorporating the changes. Table 4 shows the results.

The original version for state-machine has a much high degree of complexity when compared with agent. What is more interesting though, is the rate of change in complexity resulting from 1 additional behaviour. Complexity metric has increased by 8 for state-machine version as opposed to 1 for agent. Extrapolating this trend, one could easily conclude that the state-machine implementation would become untenable with increasing number of behaviours.

3.3 Worst Case Analysis

Cyclomatic Complexity	Risk Evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
greater than 50	untestable (very high risk)

Table 3: McCabe's cyclomatic complexity values. [9]

	McCabe's Complexity	
Model	Original	After Modification
State Machine	17	25
Agent	6	7

Table 4: Complexity of State transitions and Agent plan initiation

Intrigued by the differences in complexity between state-machine and agent implementation we proceeded to analyse the worst case scenario for both models. For state-machine any additional behaviour may be represented by a new node. Hence, for state machine with n nodes, a new behaviour may introduce up to $2n$ new transitions resulting in an additional 2n logical paths. Hence the increase in complexity for each additional behaviour is $2n$. Summing it over the total behaviours t we arrive at:

$$\sum_{n=1}^{n=t} 2n = t^2 + 1$$

With BDI agent, adding a new behaviour amounts to adding a new plan. Hence the complexity for a total of t behaviours is just t. Hence we have shown that the complexity for FSM and agent are of a different magnitude, quadratic and linear respectively. Hence we conclude that agent behaviour lends itself to more complex and intelligent systems, when compared with FSM.

4. PERFORMANCE ISSUES

To objectively compare the speed of agent implementation with that of the BDI implementation we devised the experiment below. We created a new game, where two state-machine Bombermen, played against two agent Bombermen, until game completion. The game was repeated 2000 times to balance out any unusual results. Both implementations had a similar level of intelligence defeating each other roughly equal number of times. The average cycle time for both implementations can be seen in figure 7.

The results showed that runtime-performance for both implementations are of a similar magnitude. The slightly higher cycle-time could be attributed to the additional overheads involving JACK agents.

As the games are getting more complex all the time we attempted to measure, how the runtime-performance will degrade with increasing number of characters. The experiment above was repeated with number of characters increasing

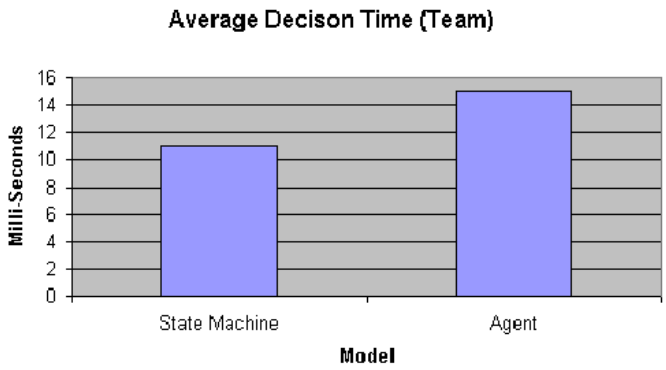

Figure 7: Performance of both models after 2000 runs in a Many Vs Many situation

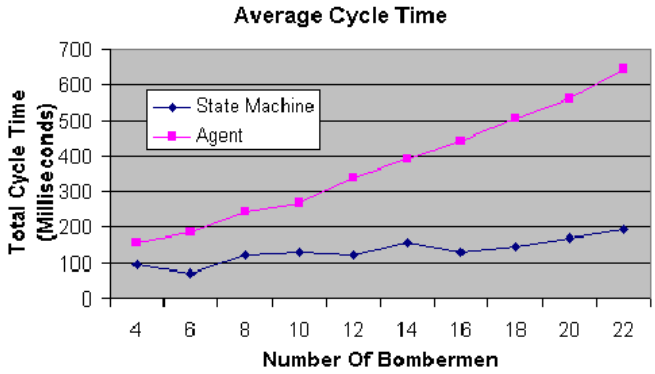

Figure 8: Performance of both models after 2000 runs, as the number of Bombermen increase by 2's from 4 to 22

from 2 to 22 in steps of 2. The results are shown in figure 8.

As can be seen, both implementations show a linear increase in cycle-time as the number of characters increase. This would allow designers to predict the run-time performance with reasonable confidence. The rate of increase however, is much greater with agents possibly due to the additional overheads involving threads and events for each agent. State machine implementation allows itself to be optimised easily when compared to BDI agents. We have proposed some hybrid designs, which may take advantage of both models. More research must be done in this area.

5. CONCLUSION

The conclusions we can draw from this research can be distilled down into two categories, design and code complexity, and run-time performance.

5.1 Complexity

- The design complexity of the state machine (quadratic) and the agent (linear) are of a different order.
- Code complexity for the state machine increases *significantly* as the game scales up with more behaviours. For the agent model, the complexity increases linearly.
- We conclude that the agent model is more suited to the

new type of *intelligent* games emerging in the market, from a complexity point of view.

5.2 Performance

- The run-time performance for FSM and agent model is comparable when the game entities involved are few.
- There is a linear increase in cycle time for both models, though the overhead is much higher for the agents. This makes FSM the natural choice when the number of entities are large and the speed is critical.

6. FURTHER WORK

6.1 Team Behaviour

Team behaviour was not thoroughly tested during this research, as time constraints prevented any significant progress. This makes team behaviour a natural extension of this research. Initial investigations showed that an agent implementation of team behaviour would not be overly difficult. Pursuing this line of research would not be possible without a state machine equivalent. It would be of particular interest to measure how the team behaviour affects complexity and performance issues.

6.2 Hybrid Approach

Some research must be done to see if a Hybrid model could be produced that take advantage of a state machines performance and an agents ease of design and coding. To our knowledge there has been little investigation into this topic. Some possible hybrids could be.

- An agent which controls the state transitions for *all* state machines. In this way only a single agent would ever be created, reducing the accumulative overhead, while taking advantage of a state machines performance and centralising its *rule based evaluations* in the agent.
- An agent definition that could only post specific events, given the evaluation of an internal state.

This is by no means an exhaustive list of configurations, yet could serve as a starting point in any research into a hybrid approach.

Acknowledgments

Special thanks must go to Peter Bertok for his support and guidance in fine tuning the direction of the research and for the benefit of his many years of experience.

We would also like to thank the Agents at RMIT group, for their indirect and direct support through the year especially James Harland, Michael Winikoff, and Lin Padgham. Thanks must also go to those who participated in the design complexity experiments, whose participation came at exactly the right moment.

We must also make a special mention of Rein Van Noppen who provided equipment during the experimental phase.

7. REFERENCES

[1] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity.

[2] R. Evans. The future of ai in games: A personal view. *Game Developer*, pages 46 – 49, Aug. 2001.

[3] C. Hecker and Z. B. Simpson. State machine aka (non)deterministic finite state machine, finite state automata, flow chart. *Game Developer*, page 8, Jan. 2001.

[4] H. R. Lewis and C. H. Papadimitriou. In *Elements of the Theory of Computation Second Edition*, pages 55–111, Prentice-Hall, inc, 1998.

[5] A. O. S. A. P. Ltd. Jack intelligent agents, practicals. 2001.

[6] A. O. S. A. P. Ltd. Jack intelligent agents, user guide. 2001.

[7] A. O. S. A. P. Ltd. Simpleteam technical brief. 2001.

[8] P. Molyneux. Postmortem: Lionhead studios' black & white. *Game Developer*, page 8, June 2001.

[9] R. S. Pressman. In *Software Engineering A Practitioner's Approach*, pages 120–125, McGraw-Hill Companies, inc, 1997.

[10] A. S. Rao and M. P. Georgeff. Modeling rational agents within a bdi-architecture. 1991.

[11] A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. 1995.

[12] M. Tambe. Executing team plans in dynamic, multi-agent domains. 1996.

[13] M. Tambe. Towards flexible teamwork. 1997.

[14] J. Thangarajah. Representation of goals in the belief-desire-intention model. 2000.

[15] D. Weinstein. State decision and consequence separation aka duplicated state decision points. *Game Developer*, page 13, Dec. 2000.

[16] S. Woodcock. Game ai: The state of the industry 2000 - 2001. it's not just art, it's engineering. *Game Developer*, pages 36 – 44, Aug. 2001.