NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SURVEY OF AVAILABLE ARTIFICIAL INTELLIGENCE TECHNOLOGIES FOR ADDITION INTO DELTA3D

by

Aaron J. Mueller

September 2006

Thesis Advisor:	Christian J. Darken
Second Reader:	Karl D. Pfeiffer

This thesis done in cooperation with the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: Survey of Available Artificial Intelligence Technologies for Addition Into Delta3D	5. FUNDING NUMBERS
6. AUTHOR(S) Aaron J. Mueller	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT (maximum 200 words)

This thesis explores the addition of Artificial Intelligence (AI) capability to the Delta3D Gaming and Simulation engine developed at the Naval Postgraduate School. A look at what types of AI capabilities exist and their potential to add value to the project is presented. This look includes the use of specific AI technologies, such as State Machines and Pathfinding, as well as the potential use of existing open source packages. One growing trend in the commercial game industry is the use of AI Middleware packages, allowing developers to buy what technologies they need and reduce development time. This thesis covers the link between AI and animation, specifically comparing how animation is handled by Delta3D and UnrealEngine. One final area covered is the use of scripting to generate behaviors within a game or simulation. Again, UnrealEngine, specifically UnrealScript, is considered as a potential model for a scripting language based on the Python programming language. Python was chosen based on its integration with the underlying C++ base code. By following the game industry's lead, one has a pool of potential options and avoids attempting to reinvent the wheel.

14. SUBJECT TERMS Artificial Intelligence, Animation, Scripting Language, Finite State Machine, Path Search Algorithm, Python, Delta3D, Virtual Environments	15. NUMBER OF PAGES 69
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SURVEY OF AVAILABLE ARTIFICIAL INTELLIGENCE TECHNOLOGIES FOR ADDITION INTO DELTA3D

Aaron J. Mueller
Lieutenant, United States Navy
B.S., University of Hartford, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND SIMULATIONS

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Lieutenant Aaron J. Mueller, U.S. Navy

Approved by: Dr. Christian J. Darken
Thesis Advisor

Dr. Karl D. Pfeiffer, MAJ, USAF
Second Reader

Dr. Rudolph P. Darken
MOVES Academic Committee Chair

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the addition of Artificial Intelligence (AI) capability to the Delta3D Gaming and Simulation engine developed at the Naval Postgraduate School. A look at what types of AI capabilities exist and their potential to add value to the project is presented. This look includes the use of specific AI technologies, such as State Machines and Pathfinding, as well as the potential use of existing open source packages. One growing trend in the commercial game industry is the use of AI Middleware packages, allowing developers to buy what technologies they need and reduce development time. This thesis covers the link between AI and animation, specifically comparing how animation is handled by Delta3D and UnrealEngine. One final area covered is the use of scripting to generate behaviors within a game or simulation. Again, UnrealEngine, specifically UnrealScript, is considered as a potential model for a scripting language based on the Python programming language. Python was chosen based on its integration with the underlying C++ base code. By following the game industry's lead, one has a pool of potential options and avoids attempting to reinvent the wheel.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

There are many people to thank for their support in the completion of this thesis. In no particular order:

Dr. Christian Darken: thesis advisor. Thank you for some excellent learning experiences in the many classes I took with you and increasing my interest in the general field of Artificial Intelligence. Thank you for your patience with me. It is my sincere hope that the results of this evaluation can be carried forward to the implementation of AI in Delta3D.

Dr. Rudy Darken: Academic Associate and current MOVES Institute Director. Thank you for being a source of incredible enthusiasm and ideas.

CDR Joseph Sullivan: Program Officer. Thank you for your mentorship and leadership during my time here.

The Delta3D team: Erik Johnson, Chris Osborn and Matt Prichard. Thank you for all the help and support, especially to Chris Osborn over the last few months. Thanks for patiently answering my questions; even the ones that were in the README file.

My amigos: Joaquin S. Correia, Louis Gutierrez, Murat Onder, and Oliver Tan. Steve, thanks for being a great roommate and putting up with me for 2 ½ years. Congratulations, you survived! Lou, thanks for all your help and psychoanalysis, but mostly your friendship. Murat, thanks for all your help early on in the curriculum. I owe a lot to you. And last but not least, Oliver. Thanks for your help during the agents class.

Prior and current MOVES students. Thanks to those who went before me and to those students who have 2, 4, or 6 quarters left to go.

Lt Col Karl Pfeiffer, USAF. Thank you for all your guidance and help.

To anyone I may have forgotten to list by name, thank you!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THESIS STATEMENT

One common feature found in popular game engines today is how believable the Artificial Intelligence (AI) of a Non Player Character (NPC) appears to the player. When the AI of a game is not believable, it is often perceived by a player as being "dumb" or unintelligent. In this situation, a player often prefers directly competing against other human players either in a Local Area Network or over the Internet.

Similarly, in a military simulation, it is often desirable to train with or against other human participants. When the training task requires multiple participants and not all are available, then some form credible AI capability is needed. Recognizing this need, it is good engineering practice to look at what other commercial game engines are doing with respect to AI. Using that as a starting point, one can define what constitutes minimum functionality, and then build from there. Or, said another way, the capabilities found in commercial game engines will form the baseline of current military requirements, as appropriate.

B. MOTIVATION

In an effort to apply entertainment type technology to military applications, the Delta3D project, a product of the Modeling, Virtual Environments and Simulations (MOVES) Institute at the Naval Postgraduate School (NPS), is building a game and simulation engine using readily available open source software tools and libraries. The goal of the project is to develop a full-featured open source gaming and simulation engine based on existing open source software. In its current version Delta3D uses the following open source projects: OpenSceneGraph for graphics and scene rendering, found at http://www.openscenegraph.org/ (OSG05); Open Dynamics Engine for physics simulation, found at http://ode.org/ (ODE05); Cal3d for character animation,

found at http://cal3d.sourceforge.net/ (Cal3d05) and OpenAL for sound, found at http://www.openal.org/ (OpenAL05). Going the open source route avoids the high costs associated with the licensing of current game engine technologies such as the Unreal Engine of Epic Games™. By making the source code available, it is hoped that a vibrant developer community will form to both support and enhance the Delta3D engine. In return, the military stands to gain a continuously updated simulation and training tool and, most importantly, one that it can freely use and repurpose for any and all applicable training needs.

The Unreal engine is one of the most successful game engines available on the market today in the first person shooter genre. Not only does Epic Games produce their game Unreal Tournament with it, but other game development houses license the Unreal engine to produce commercially available games. Some examples include Tom Clancy's Rainbow Six: Raven Shield by Ubisoft and Shadow Ops™: Red Mercury by Atari, Inc. Similarly, the Army Game Project at NPS paid for the license to use the Unreal Engine in the development of America's Army. Game engines in general are quite costly to develop, so some of the more successful ones like Unreal are used to generate additional revenue via licensing their gaming technologies.

Over the past few years, the military has grown more receptive to the use of game technology for training. More specifically, the U.S. Army created America's Army: Operations as a public relations exercise and, ultimately, using it as a potential recruiting tool. Since its release in the summer of 2002, interest in using it for training has increased. However, to repurpose America's Army for a specific training objective requires additional licensing costs and is subject to many restrictions on its potential distribution. The Delta3D project was created to break this model. In the commercial world, the object is to make money on the games created. Since the military is not selling the game or simulation and the target volume of users is much lower than the market of a popular first-person shooter game, the high licensing costs make the use of commercial gaming engines undesirable. With the Delta3D engine being freely available, the military can make many training specific applications and distribute it to any and all

commands as it sees fit. The main costs involved presently include funding of the small full-time developer team. As the engine matures more, it should gain more open source developer support and not necessarily require a huge expenditure.

The Unreal Tournament series and the Unreal Engine provide an appropriate model for comparison to the current Delta3D engine. In fact, in a document prepared for the Naval Education and Training Command (NETC) in November 2004, entitled Game Engine Comparison Table, specifically uses Unreal as a basis for comparison with the P-51 project (now called Delta3D) (Darken04). By using the Unreal engine as a basis for comparison, one can see that the Delta3D project is steadily progressing on the way to producing similar functionality but without the high costs and licensing restrictions. However, there are two specific areas that need to be developed to more closely match the functionality available with the Unreal Engine: Networking support (both local and wide area) and Artificial Intelligence (AI).

Lacking networking support in a game or training product, one is left with only a single player mode. Generally speaking, single player games hold little or no entertainment value without some form of AI to play against. In the application of game technology to training, a single player game might be most appropriate for a partial task trainer like the Delta3D Fire Fighter trainer. However, except for very small fires, it is highly likely that the trainee would be part of a fire fighting team. In extending this example to include team training, the ability to participate in a multi-participant simulation or game over either a local or wide area network is needed. Networking capability facilitates training with all the members of the team. This example represents an ideal situation; however, it might not be possible for all of the members of the team to be present. In an example where one is training against an adversary, there might not be enough human participants with the necessary expertise to play such a role. In either case, a game or simulation should have some form of credible AI to fill in where human players are not readily available. To be engaging for a player in a game or a trainee in a simulation and to invoke a sense of immersion,

a game or simulation needs the ability to accommodate multiple participants, whether they are human opponents or software-based agents that attempt to mimic realistic human behavior.

Recognizing the need to incorporate networking and AI is just a first step. This thesis will look at the existing game and simulation engines, specifically with respect to AI, and recommend some baseline requirements for the immediate future. By incorporating common AI functionality found in existing games, a more conducive environment to continually improve and create content for the Delta3D engine is possible. The addition of AI capabilities will add another level of realism beyond the visual capabilities of the existing Delta3D engine.

C. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- **Chapter II: Background.** This chapter will briefly define a game engine and then talk more about the current state of the art in the game industry and Department of Defense. It will also briefly discuss why artificial intelligence is important and highlight differences between the two domains.
- **Chapter III: Elements of a Successful Engine.** This chapter will discuss some of the common AI technologies used in today's games. It will also talk about the role of AI with respect to animation. Specifically it will address the role of scripting languages in controlling animation.
- **Chapter IV: Scripting Languages.** This chapter will look at some of the common scripting languages in use today. Then it will focus on some advantages of scripting languages and why scripting languages are suitable for AI.
- **Chapter V: Conclusions and Future Work.** This chapter will discuss some general conclusions and provide some suggestions for future research.

II. BACKGROUND

A. INTRODUCTION

In this chapter, a brief familiarization with game technology is presented. Some common terms used in the game industry are defined and a look at the current state of the art is considered. The other major theme of this chapter is to discuss the importance of AI to the game industry and the Department of Defense (DoD). Finally, a brief look at where game and DoD needs differ is provided.

B. SOME COMMON GAME ENGINE TERMINOLOGY

For those not familiar with the game industry, a brief discussion of some common game terminology is prudent. As in any technical field, the game industry comes complete with its own unique jargon. In addition, in the right context, game and simulation may be considered interchangeable terms. Unfortunately, there exists a somewhat negative connotation attached to the word “game.” However, as more game technologies are appropriately and successfully used in training and simulations, this attitude should change.

1. Game Engine

Probably the most commonly used term in this thesis is game engine. What is a game engine? More often than not, many mistake the entire game for the game engine. One possible analogy is that of a car and a car engine. It is possible to take a car engine out and build another body to put around it. Similarly, the game engine is the basic unit for non-game specific things, like the rendering system. The game engine will always handle how objects in a game are drawn to the screen. On the other hand, the game on top of the game engine is made up of content. This would include things like the models (vehicles, buildings or characters), the animations, sounds, AI and even physics. Initially it might seem like physics is something that would be part of the engine;

however, it would be possible to envision a game engine being reused for a flight simulator. In such a case, the physics of say a first person shooter type game wouldn't necessarily make sense, and instead a physics model for aerodynamics would be appropriate. (Simpson02)

2. Non Player Character

Another common term, specifically in relationship to AI, is Non Player Character, or NPC. Any character in a game that is not controlled by a human player is a Non Player Character. Often the NPC is one of the most obvious places in a game where AI manifests itself. NPCs can be either friend or foe. In a team based setting, NPCs or human players can act as members of the team. On the other hand, NPCs can also act in an adversarial role. An example might be a force on force exercise. If the goal is to capture the enemy flag, and the enemy has three team members, the game would create three NPCs to act as the adversaries and use some form of team AI for that type of game.

NPCs from any game consist of essentially two parts: a body and a controller. The body is the physical appearance of the NPC or 'bot. The term bot is also used quite often in the game world. Bot is short for robot. In a sense, the character is like a robot. Though that name may not quite adequately describe NPCs today, it is commonly used. The body could be a human form, or some type of alien life form. Furthermore, a bot is not necessarily limited to a human form. It could possibly take other common forms such as tanks, planes, or even ships. Its representation in the game is limited in part by developer imagination and part by the number of polygons that make up the model. Complex models have hundreds of thousands to millions of polygons. Having a large number of NPCs with a high polygon count, developers run the risk of limiting what the game engine can handle. This is a function of the current technology and its limitations, which as technology marches on becomes less important. The other part of the NPC is a controller. In a sense, one can think of the controller as the brain of the NPC. It is here where the AI is contained. The controller senses its environment in whatever ways the designer has called for, makes a decision on

how to act and then executes. In better-designed AI, the resulting actions hint at intelligent control. In the end, AI is still a rather difficult science problem but leaves plenty of avenues to explore.

3. Scripting Language

One possible definition of a scripting language is "any programming language that is created to simplify a complex task for a particular program." (Berger02) A scripting language in a first-person shooter game might be used to define the AI of any of the various NPCs that appear. Similar to the relationship between game and game engine, a scripting language is generally broken up into two parts: the scripting language and the scripting engine. The language is the scriptwriter's interface to the underlying scripting engine. Like almost any programming language, it has a specific syntax and structure and the scripts are run through a compiler to be translated to a format that the scripting engine can handle. The engine, also called an interpreter, takes the compiled script and executes it. This compiled script is called a bytecode stream, which contains everything needed to execute the script. More complicated scripting languages often take a long time to compile, so the compile is often performed before to reduce this overhead. (Berger02) It is this compile to bytecode process which makes it similar to the Java Virtual Machine.

C. CURRENT STATE OF THE ART ENGINES

Currently, there are several different commercial game companies that have developed their own game engines. When it comes to pushing the latest technology to the limit, the following game engines come to mind:

- Unreal Engine™ by Epic Games: Unreal Tournament series, numerous 3rd party licensees
- Quake Engine by id Software: Quake I/II/III
- Doom 3 Engine by id Software: Doom 3
- Source Engine by Valve Software: Half Life 2

- CryEngine by CryTek: Far Cry
- Torque Engine by GarageGames

Generally speaking, when the term “state of the art” is used, often the assumption is that this refers to computer graphics. Many games strive for visual realism which always pushes the latest video circuitry available on the market. As game developers quickly begin to make the most use of the available hardware, the graphics hardware manufacturers are working on the next generation of hardware and capabilities. As developers push the limits of what is currently capable, manufacturers are locked in battle to produce the newest and most capable hardware. Simply put, it creates a vicious cycle.

Over the last few years, computer graphics have been driving game development and have pushed the limits of current graphics hardware and displays. The industry has almost reached a point where computer graphics technology is at the point of diminishing returns. (Tozour02) The new frontier, or perhaps an often neglected one, is Artificial Intelligence. If one were to take a look at a typical game development company to see how it is structured, one could expect to find a lot of programmers focused on the game mechanics; level designers to develop the levels and script actions; artists to draw monsters, other characters and weapons, etcetera. From looking at comments from the AI community, game reviews from the past few years, and other general comments, many game developers have voiced the concern that not much time is spent on AI for games. This attitude is changing somewhat as some of the latest generation of games demonstrate some rather sophisticated AI. And while game companies have started to devote more human talent and resources at the problem of AI in games, there is still room for improvement. (Tozour02)

A final thought on the above mentioned game engines as it applies to this thesis is to design an engine with the capability to completely replace or augment existing AI in these games. Game Engines like Unreal Engine and the Doom 3 Engine have spawned numerous web sites dedicated to inform those who wish to modify or ‘mod’ the game. Being able to customize the existing AI or completely replace it is a valuable capability.

D. DIFFERENCES IN AI FOCUS

Up to this point, much of the focus of this thesis has been placed on the game industry. The game industry continually pushes the limits of computer hardware to ever increasing heights, specifically in the realm of computer graphics. The capabilities of the Graphics Processing Unit (GPU) have increased at a rate even higher than Moore's Law, an outside observer might suggest that the point of diminishing returns is rapidly approaching. As the industry approaches photo realistic rendering, not much value is added by successive increases in visual quality. The game may look more visually stunning, but it can still leave a player wanting. As more powerful GPUs are introduced and can handle the heavy lifting of rendering the screen, the Central Processing Unit (CPU) is available to tackle other tasks, such as AI. With limited resources in the past, when it came to AI capabilities, the developers had to resort to tricks and shortcuts to maximize the CPU cycles that were available to them. Since most of the CPU time was used for graphics or other functions, the AI presented in the game was often very limited.

In the Department of Defense, relatively little emphasis has been placed on visual realism. Simulations are often used for analysis and not necessarily used for training. In the eyes of an analyst, the visualization while the simulation is running is generally not so important. If the visual realism desired doesn't drastically increase the total time required to complete a simulation, then perhaps a simulation professional would not object to the improvement in graphics. In training however, it is often the case that visual realism can improve the participant's sense of immersion or realism. While this is not easily measurable, a common sense approach would say that suspending disbelief effectively draws the participant into the game or simulation. Similarly, to enhance a participant's level of realism or immersion, an NPC's behavior should be believable. While the game industry tends to focus first on the graphics and then the AI, the DoD has worked toward more realism in the arena of human behavior while mostly ignoring the graphics and the potential improvements it brings. (Scott02)

1. Game Industry

In the world of AI, there is generally a difference in focus of the entertainment and defense industries. Where the defense industry generally looks to realism, the game industry has focused on making things look good enough. In an interview with Chris Butcher from Bungie, makers of the popular Xbox games Halo and Halo 2, he makes an interesting observation that is important (Valdes04). He asked the interviewer and author of the article, Robert Valdes, if he was familiar with the idea of the "Uncanny Valley." When the interviewer responded he wasn't familiar, Butcher went on to explain:

> As characters become more photo-realistic, you start to believe in them more and more. With human characters, you get to a certain point of realism. What happens is there are characters that are so realistic you want to believe they are actually human. Then you notice their deficiencies. They have very plastic skin or very wooden eyes. All of the sudden they just become creepy. They are like zombie people, rather than appealing computer people. The appeal of the character rises, then drops dramatically, then rises again as you approach photo-realism.

From here, Butcher comments that the design goals for Halo and Halo 2 were not to be photorealistic. But his more pressing concern was the idea that behavior in games is approaching a similar 'uncanny valley.' In combat situations, the characters seem very lifelike with interesting behavior, both individually and in groups. However, a problem occurs when the AI is faced with situations they are not programmed for and thus do not have proper reactions. Often the reactions that result make the AI look unintelligent. But, probably the most relevant statement is the final one from the article:

> The challenge with AI in games right now is, 'What are the boundaries of AI? How do we hide those boundaries from the player? And how do we go beyond them?' That is what artificial intelligence is all about. It's about tricking the player into believing that there is something intelligent there.

This final statement is one area that seems problematic, highlighting the differing opinions between entertainment and defense. In the entertainment industry, it is sufficient to 'trick' the player into thinking there is an intelligent

player there. But for DoD purposes, it is not sufficient to trick the trainee. Many of the tricks are only applicable to limited situations and would not likely work or produce desirable results in a more general situation.

2. Department of Defense

Where games generally have seemed to focus on using scripted sequences and perfect knowledge of the game environment to cause NPCs to act, the DoD, specifically the Defense Advanced Research Projects Agency (DARPA), is looking toward Biologically-Inspired Cognitive Architectures (BICA). This is the subject of DARPA's BICA program, which is currently seeking proposals. From the DARPA Program web page:

> The goal of the Biologically-Inspired Cognitive Architectures Program via this BAA is to develop, implement and evaluate psychologically-based and neurobiologically-based theories, design principles, and architectures of human cognition. In a subsequent phase, the program has the ultimate goal of implementing computational models of human cognition that could eventually be used to simulate human behavior and approach human cognitive performance in a wide range of situations

DARPA feels that, while the traditional approach to machine intelligence pursued by the AI research community has produced many achievements, it has not quite lived up to the grand vision of "integrated, versatile, intelligent systems." With this in mind, DARPA is placing its hopes in the fields of neuroscience and cognitive psychology. Over the last several years, advances in equipment and experimentation have allowed a substantial improvement in understanding the physical structure and function of the human brain. Running in parallel to these advances are improvements in computational theories and architectures based on modeling functions of human cognition like perception, memory, decision making and even problem solving. Some of these psychologically-based models of cognition like ACT-R and SOAR have been able to yield impressive simulation of human-like behavior. By working in the context of human cognition, DARPA hopes to take a step back and take a fresh new look at the design and implementation of human cognition architectures. The goal is to dramatically

improve learning of machines via these new architectures and new theoretical work in the areas of neuroscience and cognitive psychology. The important thing to note is DARPA's definition of learning. According to the project information, DARPA defines learning as "organizing data for creative and adaptive uses." In this case, the result of learning is to be able to effectively deal with new situations. (DARPA05)

E. SUMMARY

The game industry has only recently started to put more emphasis on improving the AI capabilities in the games that are produced. In general, game AI often only exhibits some signs of intelligence. If one were to watch the AI closely, depending on the game and capabilities used, the AI only acts when it senses the player. Chris Butcher, who developed the AI for Halo and Halo 2, comments that if one were to sneak up on the NPCs in the game and then just sit and watch, the NPCs will do nothing until they sense the player. In the real world, it is conceivable to think that the NPC should be searching for his opponent, or have some other goal to achieve. For example, in a DoD application, a NPC soldier might be digging a ditch or cleaning his weapon. An exception would be in the case of an ambush. The NPC would be strategically placed to lie in waiting. Then, the NPC would come alive to engage the player. One could perceive this as intelligent behavior even though the NPC just sits there. The hard problem is what to do once the AI springs into action; how does one make it appear that there is an intelligent being where there really isn't one?

III. COMMON AI CAPABILITIES

A. INTRODUCTION

In this chapter, a look at some of the underlying technologies found in today's successful game engines is offered. What are the common AI capabilities needed for use in a game engine? These technologies will be good candidates for inclusion into Delta3D. Instead of building AI from scratch, some companies produce AI middleware. A discussion of some of the AI middleware packages available today is provided. Finally, a look at the relationship between AI and animation is considered. The end result should be a basic understanding of the common methods from the field of AI that are used currently.

B. GAME AI CAPABILITIES AND TECHNOLOGIES IN USE TODAY

Since the goal of the Delta3D project is to create an open source gaming and simulation engine, it is useful to look at what AI capabilities and techniques are in use in the more successful game engines today. A look at both current and past best selling games, like DOOM 3, Half Life 2 and Unreal Tournament will be a good introduction to some of the current technologies in use.

Generally speaking, any game companies that advertise advanced AI capabilities are generally reluctant to make such information or source code available. Those that have licenses to develop with a particular game engine like Unreal Engine or the Doom 3 Engine typically have access to developer resources, and this access comes at a cost. Despite this relative secrecy and reluctance to share ideas, there are some common techniques that are in use today. Examples include the following: Pathfinding, Dead Reckoning, State Machines, and Line of Sight. The following subsections will discuss precisely these methods listed above.

1. Pathfinding

As the name indicates, pathfinding is an algorithm to find a path through an environment. There are a handful of algorithms, but the A* (spoken like A-star) search is a method of finding the cheapest path through a particular environment. It is a directed search algorithm and a heuristic to evaluate the cost of moving along a particular path in the search space. By using this heuristic function, the amount of processing time to find a solution is minimized. In such a scheme, if movement has the equal cost, the cheapest path works out to be the shortest path. (Russel02)

In order to take advantage of this path searching capability, the game environment must be represented in such a way that specifically defines where movement is allowed. Then, by passing a start position and a goal position, the algorithm can search for the path having the least cost. Successful completion of the A* search yields a list of points representing the path. (Rabin04)

2. Dead Reckoning

Dead reckoning is way of predicting an object's future position based on its current position, velocity and acceleration. In essence, this is a relatively simple form of prediction, since most objects have movement that resembles a straight line over a short time. Some more advanced versions of dead reckoning may be able to give an idea of how far an object may have moved since it was seen last.

This technique is often an effective way to control the difficulty level in a first-person shooter type game. To decrease the difficulty level for the player, the computer's accuracy at "leading the target" when shooting is lowered. Because projectiles can't travel instantaneously, the future position of targets must be predicted. With this prediction, the NPC can aim the weapon at the point it has predicted in order to hit it. One possible way to decrease the difficulty level for the player could be to make the NPC lead the target too much, causing it to miss

more often. Conversely, to make the game more difficult, the prediction point will be made more accurate. (Rabin04)

3. State Machines

Fu and Houlette (Fu02) claim that state machines are the most widely used technique in game AI programming. Due to its broad use in the game industry, the state machine can be considered a cornerstone of game AI. A few types of state machines in use today are discussed in the following subsections.

a. Finite State Machine (FSM)

As its name implies, a finite state machine is described by a limited or 'finite' set of states and transitions. An important additional restriction is that only one state may be active at any given time. Generally, a state represents a specific behavior like Patrol or Attack. In a FSM, the state can either actively poll or passively listen for events that will cause a transition from its current state to one of the other described states. (Rabin04)

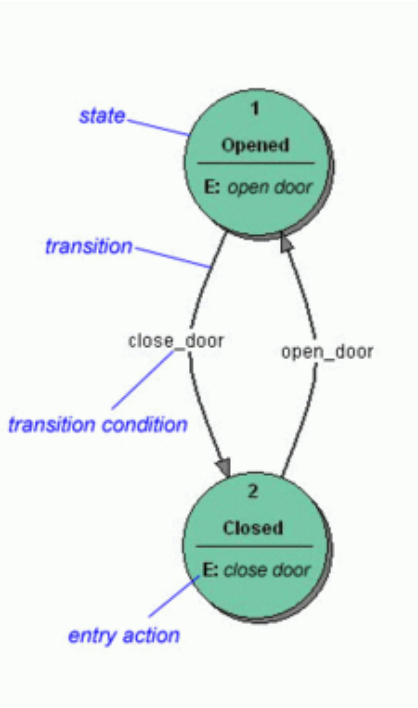

Figure 1. A Finite State Machine Example. (From: Wiki05)

b. Stack-Based State Machine

A stack-based state machine is similar to a FSM; however, it has the ability to remember past actions. The memory ability is achieved by using a common memory structure called a stack to store the past actions. The stack-based state machine was developed based on the realization that it may be useful to be able to return to a previous state. Having a memory would be useful for a goal-based AI. (Rabin04)

c. Fuzzy State Machine (FuSM)

A fuzzy state machine is similar to the FSM; however it uses the AI technique of Fuzzy logic to influence a change in state. Where a traditional FSM has a collection of fixed states and transitions and a predicable outcome, a FuSM uses weight values to influence the state transitions. The FuSM is not required to be in only one state at a time, but can be partially in one or more states. The FuSM, unlike the FSM, is non-deterministic. (Rabin04)

4. Line of Sight

When playing against other players, a human player can look at the computer screen and be able to tell if his teammates and opponents are visible. How does a NPC handle this inherent human capability? One technique is called Line of Sight (LOS). In performing LOS calculations, a ray is drawn from the observer to the target. The computer knows where all objects are and it may choose one or several points on the target to determine its visibility. In the case of a single ray, the algorithm might choose the topmost pixel and draw the ray there. If the LOS ray does not intersect some other object before reaching the target, it is considered to be visible. Depending on what other objects are in the scene, the target might be completely or only partially obscured. An extreme example might be that of a leaf in the path between the observer and target. A leaf may be pretty small and likely would not obscure the target. However, if the LOS ray were to intersect the polygon representing the leaf before it reached the target, the algorithm would consider the target to be invisible.

This is not an ideal way to determine visibility. An improved algorithm might choose several points on the target and try drawing multiple rays. While this will likely reduce the chances of a target being considered invisible because of a single leaf or small object, this introduces the potential for many more necessary calculations, as you must have **n** rays for **n** points on the target. If you have only one observer and one target, this might be acceptable. However, it is common to have multiple targets, and if there are multiple observers, that only serves to increase the number of necessary calculations to determine visibility. Because many games only dedicate a certain amount of CPU processing time to AI functionality, one can see how that much of it would be quickly used up by LOS calculations for multiple targets and observers. This would ultimately leave little CPU time for other vital AI algorithms to function. (Morgan05)

5. Scripting

Scripting is a way of defining game data or logic outside of the game's development language. In the past, the game developers have often written scripting languages from scratch. However, there has been more interest in using languages such as Python or Lua as opposed to developing one from scratch. While the popularity of scripting languages has been increasing, the use of scripting languages covers a wide spectrum from no scripting to having the entire game written in script code. Table 1 below illustrates this wide spectrum. (Rabin04) Most games can be found in the middle levels because the extreme ends of this spectrum often increase risk, time and cost.

Scripting influence spectrum	
Level 0	Hard code everything in the source language (C/C++).
Level 1	Data in files specify stats and locations of characters/objects.
Level 2	Scripted cutscene sequences (noninteractive)
Level 3	Lightweight logic specified by tools or scripts, as in a trigger system.
Level 4	Heavy logic in scripts that rely on core functions written in C/C++.
Level 5	Everything coded in scripts – full alternative language to C/C++.

Table 1. Scripting Influence Spectrum. (From: Rabin04)

C. OPEN-SOURCE AI PACKAGES

One of the goals of the Delta3D project is to use existing open-source software packages and to incorporate them in the engine. As the following section will discuss, there are open-source projects that are trying to define and build the basic AI framework necessary for game development. Specifically, there are two open-source AI projects that will be covered: the OpenAI project and Flexible Embodied Animat aRchitecture or FEAR. In addition there are Partial AI packages. The available partial AI packages provide only a small part of the required AI in a game, but that small part is often done extremely well. In addition, they tend to have a vibrant support community. For the interested reader Smith05 covers more details on a few such packages.

Unlike the partial AI packages just mentioned, the two packages above are what some might call all inclusive AI packages. Such packages attempt to include a wide number of AI disciplines. In addition, many such projects start off with a small dedicated developer community. The project leader gets hired away by a big game company and is forbidden by contract to work on any side projects. With the leader effectively removed, interest dies off and the project withers on the vine. Not all projects end this way, but it seems to be a common occurrence (Smith05).

1. OpenAI

OpenAI is considered an all inclusive AI package. It was designed to be open-ended framework and development suite (Smith05). In addition it is set to work with multiple programming languages like JAVA and C++. There were also some graphical user interface tools to help out those with less programming skill. The lofty goals of the project listed the following: “We hope to be known as the OpenGL of Artificial Intelligence.” (OpenAI05) In essence, the project sought to develop a specification for AI related tools.

The OpenAI site discusses the development of such AI tools and modules like:

- Neural Networks
- Genetic Algorithms
- Finite State Machine
- Mobile Agent System

Unfortunately, this project is no longer actively developed. The web site shows posts ending back in 2003 (OpenAI05). This seems to be a common fate with many open source efforts as those involved often have other commitments to attend to.

2. Flexible Animat Embodied aRchitecture (FEAR)

FEAR was pioneered by Alex J. Champandard and was designed to be an extensible framework to create AI agents in games. The initial efforts for a working sample focused on the use of the Quake II engine; however the architecture could be used with other programs. The FEAR project approaches the problem of AI from a different perspective. The project aims to have the framework be the basis for any type of AI could be modeled. Most commercial AI middleware packages tend to focus on one particular type of AI functionality. The FEAR architecture hopes to provide a platform to build whatever AI tools are necessary for a particular application. (Smith05)

One of the pitfalls of FEAR is that it requires the program has a fairly thorough working knowledge of AI. For the adventurous and interested parties, there are several examples that implement some of the following AI techniques: Neural Networks, path finding algorithms, and path planning algorithms. Others are available and are only limited to a programmer's knowledge of AI and how to implement it in software code.

Another area that makes FEAR extensible is the addition of an XML based language for AI development. This can allow those with little programming knowledge to gain proficiency with FEAR. Though the use of XML language makes things more flexible, it is not as easy as a GUI like those found in many commercial packages or as in OpenAI.

Since its inception in 2002 by Alex Champandard, he accomplished much to improve the state of AI in games by striving toward a unified framework and toolset, the project has not seen further development since 2004. After being hired by Rock Star Studios to work on the AI for the Max Payne series of games, he had to distance himself from this project somewhat. He did however have time to write a book AI Game Development which has some examples that could be used in any games using FEAR. (Smith05)

D. AI MIDDLEWARE

One of the newer ideas in AI software is something called AI middleware. AI middleware is a software service for a game engine that performs the AI function. Basically, the game developer can buy AI middleware instead of building up an AI library from scratch, or more commonly referred to as "rolling your own." Some reasons for using AI middleware in game development might be: lack of developer experience with AI, lack of expertise to design and implement AI algorithms, tight project schedules that don't permit the in house developers enough time to develop the AI to a desired level, or the AI middleware might already contain the level of AI functionality desired for the project. (Dybsand03)

While the idea of simply purchasing the AI functionality needed for a game or simulation seems enticing, there are some potential pitfalls to consider. First, from a strictly programmer perspective, is the idea of the AI as "not invented here." A game developer often makes claims on the technologies implemented in a game or game engine, so a good AI in a game is a source of developer pride. Second, and closely related to the "not invented here" syndrome, is the idea that the developer does not have complete control over the software. Thirdly, the idea that there is a potential performance hit imposed by accessing the AI middleware library. This may also create anxiety because the developer cannot necessarily access the software code to optimize it for a particular application. Another concern is that the middleware may not provide all of the desired functionality for a given project. This would likely mean that a developer might spend much more time trying to add the functionality or looking for other middleware that more closely supports the desired capability. And finally, it is quite possible that the learning curve to implement the AI middleware into the game engine is too steep. This may end up meaning that the cost of the middleware combined with the difficulty in implementing the library is more that the time involved in writing the AI functionality from scratch.

An article in Gamasutra by Eric Dybsand takes a closer look at four AI middleware products that offer character behavior. To assess the products, the following three questions are asked:

- Question 1: What does the product do for the developer?
- Question 2: What are the main features?
- Question 3: How is the product implemented in a game?

The following are the names of the four AI middleware packages that were investigated in the article. There are other AI middleware products, but the author chose to focus on ones that offer character behavior.

- AI.implant, by BioGraphic Technologies of Montreal, Canada
- DirectIA, by Mathematiques Appliquees S.A. of Paris, France
- RenderWare AI, by Criterion Software Inc. of Austin, Texas
- SimBionic, by Slotter Henke of San Mateo, California

The article is actually split up into several individual articles; one dedicated to each of the four AI middleware offerings. Figure 2 below shows the basic relationship of character behavior to the game engine and to the middleware, while Table 2 on the following page summarizes the findings.

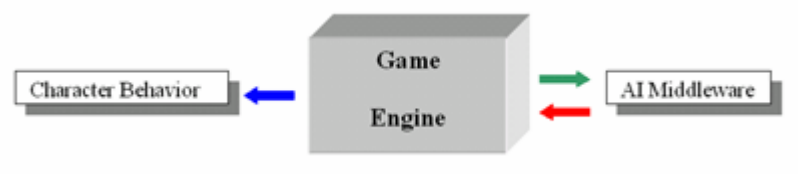

Figure 2. AI Middleware Block Diagram (From: Dybsand03)

	AI.implant	DirectIA	RenderWare AI	SimBionic
Question 1	• Tools to manage crowds • Interface between game and middleware	• Agent-based behavioral modeling	• SDK to design and develop the	• State oriented framework to define objects that exhibit behavior
Question 2	• Includes plug-ins two of the most widely used modeling packages • Hierarchical pathfinding • Rule-based decisions • Crowd tools • Assignable AI	• Real-time decision and action behavior modeling tools • Communication between agents • Behavior engine • GUI testing environment	• 3D pathfinding service • Entity management through an Entity API • XML based file configurations	• Descriptors and declarations
Question 3	• Plug-ins • SDK including support for Windows, Xbox, Playstation 2 and Gamecube	• Script and parameter files at load time	• Part of the RenderWare platform which includes Physics, Graphics, Sound and AI	• SimBionic Visual Editor • Communications such as group messaging and virtual blackboards
Misc	Target is complex animation and character control needs	More appropriate for programmer/designer than for level designers	Capable of sophisticated behavior through Kynogon AI Modules, high learning curve	

Table 2. AI Middleware Comparison Matrix compiled from (From: Dybsand03)

E. AI AND ANIMATION

This section will look at the relationship between AI and animation. Specifically it will address the role of scripting languages in controlling animation. A brief look at how the UnrealEngine handles animation is covered. Finally, a look at how animation control is implemented currently in Delta3D is provided.

When one thinks of AI, one might not think of animation as being related. The converse is also true: thinking of animation doesn't necessarily lead one to think of AI. However, upon closer consideration, the two are related and it is important to be mindful of this fact. Recalling the definition of a NPC mentioned in the previous chapter, there are two logical parts to a NPC: the controller and the body. It is here where the idea of the relationship between AI and animation begins to make sense. Simply put, the controller is the AI portion of the NPC and the body is the "physical" representation of the NPC. It is the body that gets animated. So, as the controller is sensing information about the environment and making decisions, the net result is some form of action by the body. Part of the controller's function is to decide which animation to play based on the current state.

1. Animation in Unreal

First a look at how animation is controlled in Unreal Tournament is needed. Unreal Engine has a skeletal animation system. There are other types of animation systems, like keyframe animation. The skeletal animation system has become the common technology used today in games. Unreal Engine has a set of basic animations which can be called depending on the state of the AI. If the NPC is running, it will play a running animation. If the NPC is turning to face an opponent, it has a turning animation. In addition, to transition between animations there is a way to play blend animations together. One of the benefits of the skeletal animation system is that different animations can be combined on the fly. This reduces the number of animations that need to be developed.

To get an idea of how Unreal works, a look at some code from one of the Unreal bot classes is useful.

```
state Flee
{
ignores SeePlayer, TakeDamage, HearNoise, Falling, Bump;
Begin:
       Destination = Location + 10*CollisionRadius*Normal(Location-
Target.Location);
       TweenToRunning(0.1);
       PlayRunning();
       MoveTo(Destination);
       Acceleration = vect(0,0,0);
       GotoStateLogged('StartUp');
```

Figure 3. Code Snippet from NPSbot-Flee from MV4025 Course Materials

The above code represents one of the states for the NPC in Unreal called **Flee**. As the name suggests, when a NPC transitions to the **Flee** state it will execute the code defined in the UnrealScript code. In this case, it calls a function called *TweenToRunning* to transition between the current animation and the running animation. So, most likely the bot is either standing around idle or walking. The *TweenToRunning* function allows the animation system to attempt a smooth transition from idle to run or walk to run. Now that the animations are playing, the script calls the *MoveTo* function. The *MoveTo* function takes an argument that represents the desired location for the NPC to move to. The script roughly calls for the NPC to move away from the enemy about 10 times the distance between itself and the enemy. At that point, the script instructs the NPC to alter its state. The end result of the script would be that the NPC runs away from the enemy. The state, more or less, determines what animation or animations are played.

2. Animation in Delta3D

In the Delta3D project, animation is handled with two products: Character Animation Library (Cal3D) and ReplicantBody. Since the underlying graphical capability of Delta3D is OpenSceneGraph (OSG), the project had to pick existing open source software packages that worked with OSG. Cal3d was the package chosen. It is a skeletal based 3D character animation library. It consists of two

parts: the C++ library and an exporter. The exporter is used to take 3D character models built in popular modeling packages, like Discreet's 3D Studio Max or Alias Maya, and convert them into files in a format understood by Cal3d. (Cal3d05)

In addition, the ReplicantBody software was chosen. ReplicantBody allows movement based on characters feet, ground following (clamping the character to ground) and the ability to blend animations together. (RBody05) In theory, the ReplicantBody software would allow the playing of two completely separate animations like walking and waving. Because it is based on a skeleton, the bones are grouped and therefore influence each other. So, as long as the walking and waving animations are set up properly, the two should be able to be played at the same time. This scenario would require that the bones used in the hand wave animation are not animated in the walking animation.

ReplicantBody is basically an enhancement to the capabilities to Cal3d. By setting up the .rbody file one can define the animations that the character can perform. The rbody file is a configuration file, allowing one to link the skeleton file to the materials and meshes that make up the character. On the following page located in Figure 4 is a snippet of code from the Marine.rbody file found in the Delta3D project. Using a skeleton, meshes and basic animations modeled in 3dsmax a Marine character was created for use in the demo applications available for Delta3D.

The code shown on the following page sets up a character instance. The first thing set is the path to where the supporting files are, like the skeleton file. It also defines a default action, or animation, to be played. In addition the scale of the character is set as well as an offset for the feet. As mentioned earlier, ReplicantBody offers movement based on a characters feet. Here one sees that the contact bones are what represent the feet, and that both a left and right foot are defined. A direction bone is also identified. This bone allows one to get the direction that the model is facing; in this model, it is the head bone that is used for direction. Next the animations are defined. In this sample code, two of

several animations are presented. In this case there's an animation for stand and for walk. Each animation has some parameters assigned to it like: a name, the filename of the Cal3d animation file, an action name, an action weight, a speed and a Boolean value to determine whether or not the animation will loop.

```
# ReplicantBody v0.1

	path "./marine"
	skeleton "marine_skel.csf"
	default_act "ACT_STAND"
	scale 0.026
	foot_offset 3.0 // Before scaling

	// Bones used when calculating character speed
	ContactBones {
		bone_name "Bip01 L Foot"
		bone_name "Bip01 R Foot"
	}

	// Bones from which you can get the direction
	DirectionBones {
		head_bone "Bip01 Head"
	}

	Animation {
		name "stand"
		filename "marine_stand.caf"
		act_name "ACT_STAND"
		act_weight 1.0
		speed 0.0
		is_looped 1
	}

	Animation {
		name "walk"
		filename "marine_walk.caf"
		act_name "ACT_WALK"
		act_weight 1.0
		speed 2.0
		is_looped 1
	}
```

Figure 4. Code Snippet from marine.rbody from Delta3D/data/marine/

F. SUMMARY

In this chapter, some of the common types of AI technologies in use today were presented in brief. In addition, the topic of Open Source packages available to day was addressed with a comparison of two specific packages. AI Middleware is another possibility and was looked at, specifically discussing four

separate packages available on the market today. And finally, a look at how AI and animation are important in and of themselves, but that they are interconnected as AI will drive the appropriate animation to be displayed.

IV. SCRIPTING LANGUAGES

A. INTRODUCTION

This chapter will look at some of the common scripting languages in use today. Then it will focus on some advantages of scripting languages and how this relates to AI. Finally, the application of scripting languages to a game engine is looked at. Specifically, a look at how Delta3D incorporated the Python language in Delta3D and some example Python scripts will be provided.

B. PROGRAMMING LANGUAGE BACKGROUND

Before getting into the topic of scripting languages, a brief background of programming languages in general may be helpful. A discussion of low level, high level and very high level languages follows. In addition, a look at the difference between interpreted versus compiled languages is covered. Finally, a comparison of statically versus dynamically typed languages is presented.

In the early days of computing, a programmer required detailed knowledge about the internal workings of the specific computer they wished to program. In essence, the programmer was required to know machine language and the programs were written in this low level language. Next came assembly language, which is one step higher than machine language. It is considered one step higher than machine language because the native machine language commands are replaced by mnemonic commands on a one-to-one basis. The mnemonic command names are more human friendly, making it easier to program. By running the assembly language through an assembler program, the mnemonic commands are converted into the corresponding machine language commands. The mnemonics also allow symbolic addresses for data items and the assembler program assigns them to machine addresses and ensures that there is no overlap or overwrites. Early on, assembly was used often for games found on platforms like MS-DOS or Apple; however, as games grew in size, programmers found out rather quickly that assembly language did not scale well.

Put another way, assembly language became more difficult to maintain and was not portable. The end result is that programming in assembly language is avoided except in cases were optimal performance is required, like in a device driver. (Gutschmidt04)

Next there came high-level languages. Some common examples of high-level languages are BASIC, COBOL, FORTRAN and C/C++. These high-level languages are closer in resemblance to common language and therefore easier to use. Another benefit of high-level languages is that a programmer need not have a detailed knowledge of the low level internal workings of a computer to program these instructions. Writing one line of high-level language code, an instruction, will typically translate to several machine code instructions that can be compiled or interpreted into machine code. (Gutschmidt04)

Another important distinction among computer programming languages is the idea of an interpreted versus a compiled language. An interpreted language will translate a programmer's code step-by-step at runtime. A compiled language must translate the programmer's code before it can be run. This is the compiling process. First the code is run through a compiler program to generate object code. Then, the object code is linked to any needed libraries by a linker program to produce a final executable. See Figure 5 below. (Gutschmidt04)

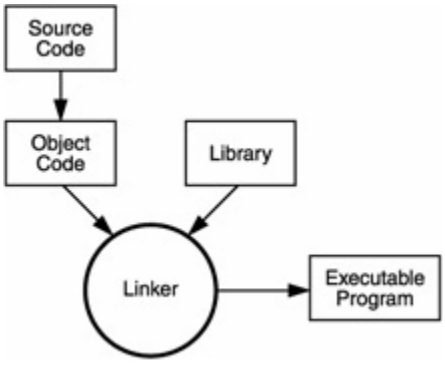

Figure 5. Compiling Process taken from Game Programming with Python, Lua, and Ruby (From: Gutschmidt04)

Finally, programming languages can be statically or dynamically typed. In a programming language, a type system defines the way in which data is organized. Statically typed languages need predefined types for any data to be used in the program. Some examples of data types are strings, integers, float, or vectors. To perform an addition operation on two integers, one might declare three variables: X, Y and Z. If the equation that defines the addition operation were Z = X + Y, all three variables must be explicitly defined as integers. It would not make sense to define the data type for the result (Z) to be a string when the code is trying to add two integers (X and Y). On the other hand, dynamically typed languages handle the data types automatically. Using the same addition example Z = X + Y, if a programmer set X = 1 and Y = 2 the language knows that 1 and 2 are integers. Furthermore, it could gracefully handle the following similar case. Say X = 1 (an integer) and Y = 2.5 (a decimal or float value). The language is smart enough to make X = 1.0 (a float) so that it can add X and Y and yield the result of 2.5. The variable Z automatically becomes a float. This automatic type capability is quite powerful, but it does cause some potential performance loss. Computer hardware of today is very fast, so this performance hit may be negligible; however, it is important to be aware of this. (Gutschmidt04)

C. COMMON LANGUAGES IN USE TODAY

As mentioned in Chapter 1 of this thesis, Berger02 defines a scripting language as “any programming language that is created to simplify any complex task for a particular program.” A look at the major game engines available today, most, if not all, have an associated scripting language. While many game companies have created languages from scratch, some are using preexisting languages to serve this function. Languages like Python and Lua are examples of preexisting languages while UnrealScript is an example of one that was created from the ground up specifically for use in a game engine. There are many scripting languages available today; however, for the purposes of discussion only three will be covered: Python, Lua and UnrealScript.

Before discussing the individual languages, it is worthwhile to note that, even today, a universally accepted definition of a pure scripting language still does not exist. Despite this, if one chose any scripting language today, the following features are commonly found:

- Interpreted language
- Simple syntax
- No pointers
- Memory allocation handled by the language
- Garbage collection handled by the language
- Interactive: can provide feedback to the programmer while running
- Code stored as plain text

Several other common features exist, but the above list is a fair representation of scripting languages. (Gutschmidt04)

1. Python

One of the most popular computer programming languages today is the Python scripting language. Developed by Guido van Rossum at the National Institute for Mathematics and Computer Science in the Netherlands, Python is copyrighted but the source code is freely available. For the trivia minded, the language actually derived its name from the TV series Monty Python's Flying Circus. It is a high-level, interpreted language and is often considered interactive and an object-oriented scripting language. (Gutschmidt04)

Some features that make it quite attractive are the following:

- Broad standard library widely available on many platforms (UNIX, Windows and Macintosh)
- Supports an interactive mode allowing interactive testing
- High portability (interpreters available for many major platforms)
- Support for Object Oriented Programming (OOP), specifically with things like multiple inheritance, classes, namespaces, etc.
- Easy integration with C, C++, Java and other languages

2. Lua

Another popular scripting language is Lua (pronounced LOO-ah). It is a lightweight language that was originally designed to extend applications, but it can be and often is used as a stand-alone language. Like Python, it is dynamically typed and has features like automatic memory management and garbage collection. It is features like this that make Lua a great language for rapid prototyping. Lua was developed by a team at Tecgraf, the Computer Graphics Technology Group at Pontifical Catholic University in Rio de Janeiro, Brazil. Like Python it is freely available, but Tecgraf retains the copyright. (Gutschmidt04)

Some features that make this attractive are the following:

- Simple syntax
- Support for OOP, specifically classes and inheritance
- Ability to extend the language in unconventional ways
- Programs compile to byte-code and then interpreted similar to the JAVA virtual machine

3. UnrealScript

Following the “build your own” approach, Epic Games spent considerable effort in making UnrealScript. UnrealScript is very similar in syntax to Java or C++ so it is should be somewhat familiar to programmers with C/C++ experience. Unlike Python and Lua, UnrealScript is a statically typed language. Variables in UnrealScript must be defined specifically and cannot change type once defined. Though the dynamic typing of languages like Python or Lua is certainly convenient, UnrealScript, with its static typing, has been used quite successfully to modify the Unreal AI capabilities.

D. APPLICATION TO A GAME ENGINE

The following section will look at two examples of how a scripting language is implemented. Specifically, a look at some sample code and explanations of what the code does will be provided for UnrealScript in UnrealEngine and Python in Delta3D.

1. UnrealScript in Unreal Engine

Since the model in Unreal is that of a State Machine, the code in Figure 6 (following page) represents a state called “Flee.” When a NPC takes damage from a bullet, it wouldn’t make a lot of sense for the NPC to just stand there and wait for another bullet to come along. A player observing this would most likely consider the AI to be unintelligent. What is more believable is having the NPC running away in response to taking a bullet. In this case, the Flee state accomplishes this. In this code, some functions are being deliberately ignored. In addition, *TakeDamage* is a function that already exists for the developer, but *TakeDamage* was rewritten to transition to the Flee state. Having functions with the same names is similar to the idea of overloading a function in C++. Again, such capability should feel quite familiar to programmers with C++ experience.

UnrealScript makes modifying the AI of NPCs in the Unreal Engine fairly easy. The following is a brief sample of UnrealScript code representing the Flee state mentioned in the paragraphs above:

```
state Flee
{
ignores SeePlayer, TakeDamage, HearNoise, Falling, Bump;
Begin:
       Destination = Location + 10*CollisionRadius*Normal(Location-
Target.Location);
       TweenToRunning(0.1);
       PlayRunning();
       MoveTo(Destination);
       Acceleration = vect(0,0,0);
       GotoStateLogged('StartUp');
```

Figure 6. Code Snippet from NPSbot-Flee from MV4025 Course Materials

2. Python in Delta3D

Selecting a scripting language that suits one's purposes is not an easy task. In the pursuit of adding scripting ability, the Delta3D project looked at two main language options: Python and Lua. After much discussion and weighing the pros and cons of each language, Python was chosen as the language to use for scripting in Delta3D. Though the selection of Python seems to be a simple choice, it was not trivial to embed the capability into the Delta3D engine. The following is a simple Python script that will create a small Delta3D application.

a. Sample Python Script

The following code, shown as Figure 7, is a relatively simple script that generates a small Delta3D application. A step by step discussion of what this script represents is necessary. Following that will be a discussion of the pieces involved to make this script perform something useful.

```
from dtCore import *
from dtABC import *

from math import *
from time import *

def radians(v):
   return v * pi/180

class TestPythonApplication(Application):
    def Config(self):
        Application.Config(self)
        SetDataFilePathList('../../data');
        self.helo = Object('UH-1N')
        self.helo.LoadFile('UH-1N/UH-1N.ive')
        self.GetScene().AddDrawable(self.helo)
        self.transform = Transform()
        self.angle = 0.0

    def PreFrame(self, deltaFrameTime):
        self.transform.Set(40*cos(radians(self.angle)),
                           100 + 40*sin(radians(self.angle)),
                           0, self.angle, 0, -45)
        self.helo.SetTransform(self.transform)
        self.angle += 45*deltaFrameTime

testPythonApp = TestPythonApplication('config.xml')

testPythonApp.Config()
testPythonApp.Run()
```

Figure 7. Code Snippet from testpython.py

The script starts of by importing functions available in other packages. Notice the asterisk ("*") on the first line. The asterisk tells the interpreter to load all functions that are in the package dtCore, the most basic part of the Delta3D engine. Simply put, these functions comprise the core of Delta3D functionality. Next it pulls in all the functions in dtABC package. The ABC in dtABC stands for Application Base Class. The dtABC class in Delta3D contains the basic application functionality. Similarly, it imports all the functions available in the math and time packages. Failure to include the math package in Python would cause errors to occur the first time a function such as the cosine ("cos") was called in the script. Since it is quite common to call upon other packages in Python, it is important to make sure that all the necessary packages are linked. This is done via the **import** command.

Next a simple function called **radians** is defined. This function is used in the PreFrame section of code. It is simply a convenience for the programmer, as the conversion of degrees to radians occurs several times in this short script file. Rather than repeatedly type the degrees to radians conversion throughout the code, the function simplifies the script and makes it more readable.

The next section of code is a class definition for the file. In this case, a class is created that inherits from the Application class in Delta3D. In this TestPythonApplication class, we have two methods or functions. The first is Config, which sets up the configuration of the small application. Because it inherits from the Application class in dtABC, it can call the Config method from its parent class (similar to the super operator in C++). The local Config method also tells the application where the data files are located. Next it creates a Python object. This object loads a model from the ones available in Delta3D as part of the download. In this particular example, the test application is using a model of a UH-1N helicopter. The AddDrawable() method is called to add the object to the scene. Finally, it declares a transform which allows the model to be manipulated in the scene.

The next block of code overloads the function PreFrame. In this case, the PreFrame function contains some code to manipulate the transform in the scene. The end result is a helicopter that flies around in a tight circle. A static screen shot is presented on the next page in Figure 8.

The final three lines of code form the basic script. The first line names a variable that creates an instance of the TestPythonApplication class defined in the file. Next it calls the Config function which configures the application and builds the basic scene by loading the model and setting the transform. Finally the Run() command is called. This starts Python running and ultimately causes the application to be generated and displayed on screen. The end result is the helicopter flying around in a tight circle within the application window. As mentioned above, the resulting application is shown in Figure 8.

Figure 8. Resulting Application from Python Script

b. Integrating Python

Although this is a fairly simple script file, there is a lot going on behind the scenes to make this script work. This section will present a background on what supporting technologies are required to link the Python scripting language to the underlying Delta3D C++ engine code.

As mentioned earlier, integrating the Python language into Delta3D is not trivial. It is not as if one can simply install Python and then link to it directly from Delta3D. To facilitate this linking requires a C++ library called

Boost.Python. The primary goal of Boost.Python is to expose C++ classes and functions to Python, which ultimately allows direct manipulation of C++ objects (in this case, Delta3D objects) from Python. Through the Boost.Python libraries, C++ code called a wrapping can be created. In most cases it turns out to be a one for one mapping of functions and methods from C++ to Python. In Delta3D, the wrappings are called Python bindings. For all the major pieces of Delta3D that will be exposed to Python, a Python bindings file must be created. On the following page is a snippet from the Python bindings file for the Delta3D character class, dtChar (see Figure 9).

```
      .def("GetVelocity", &Character::GetVelocity)
      .def("ExecuteAction", &Character::ExecuteAction, EA_overloads())
      .def("ExecuteActionWithSpeed", &Character::ExecuteActionWithSpeed,
            EAWS_overloads())
```

Figure 9. Sample Python Bindings

```
        /**
         * Returns the current walk/run velocity of this character.
         *
         * @return the current walk/run velocity
         */
        float GetVelocity() const;

        /**
         * Executes a character action.
         *
         * @param action the name of the action to execute
         * @param priority whether or not the action is high-priority
         * @param force whether or not to force the action
         */
        void ExecuteAction(std::string name,
                           bool priority = true,
                           bool force = false);

        /**
         * Executes a character action with a speed parameter.
         *
         * @param name the name of the action to execute
         * @param speed the speed at which to execute the action
         * @param priority whether or not the action is high-priority
         * @param force whether or not to force the action
         */
        void ExecuteActionWithSpeed(std::string name,
                                    double speed,
                                    bool priority = true,
                                    bool force = false);
```

Figure 10. Code Snippet from character.h

In Figure 10 above, the GetVelocity(), ExecuteAction() and ExecuteActionWithSpeed() functions are shown. These lines are a small subset of the C++ header file, character.h. Looking at Figures 9 and 10 together, one can now see that the Python function names are the same as the function names in the character.h file.

Not immediately apparent is what EA_overloads() and EAWS_overloads represent in the above code. The purpose of the syntax above is to let Python know how many arguments to expect for the ExecuteAction() function. In the case of ExecuteAction(), a minimum of one argument is required and a maximum of three is allowed. At a minimum, this function requires the input of a text string which represents the name of the action to execute. In addition it is expecting two Boolean values. If the code passes only one argument, the two Boolean values are assigned their default values as designated in the original C++ code. Similarly, the second function requires a minimum of two arguments and a maximum of four. Failing to present the minimum arguments will result in an error since there are no default values for the name of the action or its speed. For the interested reader, the entire file will be available as an appendix or it can be found in the Delta3D source code which can be found at the Delta3D project page (Delta3D05). Depending on where Delta3D has been installed, the characterbindings.cpp file can be found in C:\delta3d\src\python or C:\Program Files\delta3d\src\python.

With the wrapper code written, now one can call the C++ functions directly from a Python script. Continuing to use the Delta3D character bindings file as an example, it is now worthwhile to look at a sample Python script which makes use of the above wrapping. The following script file, called **animate.py** will be used to help tie the prior concepts together (see Figure 11). This script is run by running a short test application build with Visual Studio .NET 2003. The application, called **testPythonChar.exe** creates a simple scene in Delta3D and an instance of the Python interpreter. Because the Python interpreter is interactive, it allows the running of a script in real time. In the script one

character runs in a circle while the other is walking in a circle. At the same time, the helicopter flies overhead in a tight circle. Figure 12 on page 43 shows the results of running the sample C++ application.

```
from dtCore import *
from dtChar import *
from math import *
from time import *

plane = Object.GetInstance('helo')
bob = Character.GetInstance('bob')
dave = Character.GetInstance('dave')

transform = Transform()

angle = 0

def radians(v):
   return v * pi/180

while True:
   transform.Set(40*cos(radians(angle)),
                 100 + 40*sin(radians(angle)),
                 20, angle, 0, -45)
   plane.SetTransform(transform)
   bob.SetVelocity(0.5)
   bob.SetRotation(angle)
   bob.ExecuteAction("walk1")
   dave.SetVelocity(30.0)
   dave.SetRotation(-angle)
   dave.ExecuteAction("run")
   sleep(0.01)
   angle += 0.45
```

Figure 11. Code from animate.py

This simple application creates a scene in Delta3D and also invokes a Python interpreter. The interpreter is similar in some ways to the console found on some common games like Unreal or Doom. For those who may be unfamiliar with the term, one can think of the console to be similar to opening a DOS command prompt within Microsoft Windows™. The console can be used for both input and output. It can be a useful tool to help debug because it can display error messages. It can also be used to load additional functionality into the game. Similarly, it is using this Python interpreter that one can introduce some new behavior. Figure 13 on the following page shows a snapshot of the

result of typing the command **execfile('animate.py')** into the interpreter. This command is what the Python language uses to execute a script file.

Figure 12. Running the TestPythonChar Application

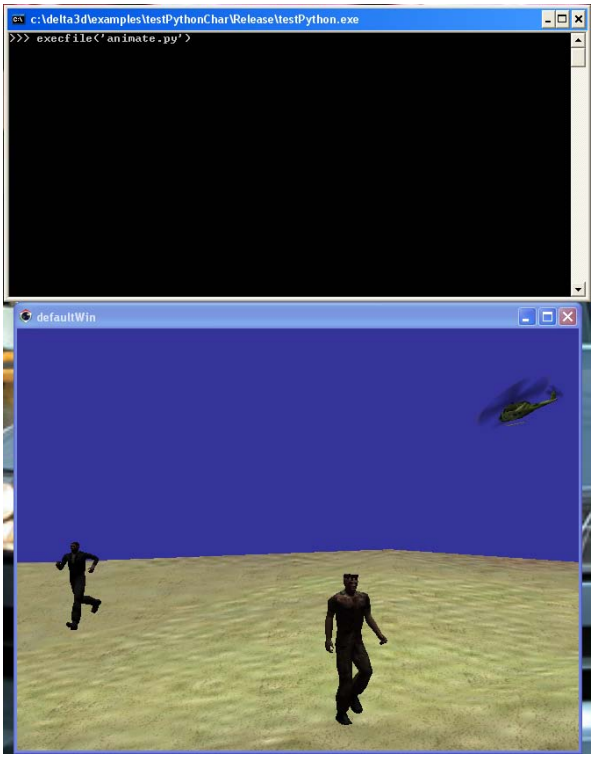

Figure 13. Running the animate.py Python Script

E. SUMMARY

This chapter took a look at scripting languages, with specific focus on UnrealScript and Python. By looking at the way Python script is handled within Delta3D, one gains a better appreciation for the capabilities a scripting language brings to the engine. Because Python supports rapid prototyping, the scripter can test out new AI behaviors as they are modified or made available. Ultimately, the engine does not necessarily need to be recompiled each time to test out a new function or behavior.

In addition, by showing that Delta3D, using the Boost.Python libraries, is capable of creating applications. As the engine matures, it is possible to conceive of a programmer using Python to build entire applications based on existing pieces of Delta3D. It is this flexibility and cross platform support, that makes Python a suitable choice to create the functionality needed. Though there is much work to be done, the capabilities that Python brings are promising.

V. CONCLUSIONS AND FUTURE WORK

A. INTRODUCTION

In this chapter, a summary of some of the things learned from looking at current game engines and available technologies is presented. In addition, some recommendations for future work are provided.

B. CONCLUSION

Until recently, the game industry has placed much of the focus of the development cycle on lavish computer graphics to increase the visual realism of their games. As was mentioned in the interview with Chris Butcher from Bungie (Valdes04), the graphical realism has achieved a point where it is not adding much to realism compared to the amount of computer resources it uses. Instead of continuing down a path of diminishing returns, the focus should change to an area that is in need of continued development: AI. Whereas before, many game development studios poured their resources into graphics and other areas of the game, AI has begun to take a larger role in determining a game's success in the market. It is not enough to simply have graphics to create the sense of immersion; the sense of immersion should increase if the participant feels as if participating against some form of intelligence. The elusive part, to this point, is figuring out how to measure a person's level of immersion.

Some of the factors that have contributed to poor AI in the past are the following: game developers not taking AI seriously and making it a last minute rush job; hardware constraints, like CPU cycles; lack of appreciation of the nature of game AI or inadequate understanding of how to apply AI techniques. In the past, programmers did work on all areas of AI within a single game. Today there is now some specialization occurring. Specifically, there are more dedicated AI programmers on a given development team. What dedicated means in this instance is that, from the start to the finish, the programmer is focused on AI. (Tozour02)

Delta3D has thus far shown that it is quite versatile based on the different types of sample environments and applications built to date. From inside a ship space in the Fire Fighter Trainer (First Person Shooter perspective) to the FAA helicopter trainer (Flight simulator), Delta3D can handle both indoor and outdoor environments. However, none of the work done so far has done much in investigating the use of AI in the application. For the Fire Fighter Trainer, an expert system might be useful for the after action review to evaluate the participant's performance in the simulation.

To recap, a final recollection of some of the technologies discussed throughout this thesis is now presented along with some specific recommendations about how one might incorporate them into the Delta3D gaming and simulation engine.

1. Common AI Technologies

In Chapter III, some of the common technologies used by commercial game engines were covered. Since Delta3D does not have any AI capability at the moment, some or all of the listed capabilities should be prioritized and added to the engine.

a. State Machine

Since it is the most common AI capability, the first one should be the implementation of the State Machine; specifically, the Finite State Machine should be the first to be implemented. Since the FSM is widely used, it will be most familiar to anyone attempting to develop a game. This could be written in the Python script language and later rolled into the C++ libraries in Delta3D if speed becomes an issue. A FSM should be a basic building block in the AI capabilities of Delta3D and is probably one of the smaller problems to tackle.

b. Pathfinding

Another common AI tool in games is the use of waypoints in pathfinding. Anyone doing modifications to Unreal or Half Life 2 has likely used a level editor to place waypoints for the AI to follow. One fellow NPS student had done some work with automatic waypoint generation. By combining the placement of waypoints with some simple line of sight checking, the algorithm figures out which waypoints are accessible from each individual waypoint. Then, once this information is available, the shortest path can be found using the A* search. Such a capability needs to be added into the Delta3D API. Though it is certainly possible to write this in Python, the A* search will likely be faster if it is written in C++ and made available as part of the Delta3D API. Ultimately, this is a good candidate for the basic AI functionality that is required.

c. Scripting

The Python language has already been selected for scripting. Much of the desired AI functionality can take advantage of Python and its ability for rapid prototyping. New AI capabilities can be quickly tested and changed in Python, potentially reducing the development cycle. Then, as mentioned in the previous two capabilities, where speed becomes an issue, the desired Python code can be transferred into C++.

2. Open-Source AI Packages

There are two open-source engines in particular looked at in this thesis. Past research had indicated that both projects were inactive based on date and time stamps on their respective SourceForge project pages. However, the FEAR project has just posted an updated Software Developer Kit (SDK) recently (Smith05). The OpenAI project has not had any active updates since early last year, while the FEAR project has recently received a new lease on life. Since the FEAR project is based on the use of the Quake Engine, the Delta3D team would have to look at the SDK as it is today with the Quake examples and write their own interface to the FEAR APIs. There is also a question of licensing, where the

FEAR project uses the General Public License or GPL, Delta3D uses the Limited General Public License, or LGPL. This is not such a problem as the developer has said that he can grant licenses in LGPL as needed upon contacting him. Both projects would be worth looking at for ideas on “rolling our own” but neither can be recommended wholesale.

3. AI Middleware

Another area mentioned in Chapter III was AI Middleware. For some game development studios with limited personnel, limited financial resources and short deadlines, using AI Middleware is certainly a viable option. Because there are licensing costs and restrictions involved, and since the goal of Delta3D is to be open source and freely available throughout the DoD, it is recommended that such AI Middleware be avoided at this time other than for a point of reference for possible methods of implementation.

4. AI and Animation

At the moment, ReplicantBody and Cal3d appear to be a good fit. As the AI develops through Python scripting, more animations to play with will be beneficial. There are only a few basic animations available to the Marine and Opfor characters currently available in Delta3D. More animations are needed to test out the blend capabilities to mix animations. Having exporters for the major modeling applications such as Alias MAYA and Discreet’s 3D Studio Max will help reduce time to produce new models for use in Delta3D applications (Smith05). As time progresses, improvements in the exporters should allow modelers to focus on creating model content and animations. Ultimately, this would improve the number of models available for use in Delta3D and create a larger library of animations. In the end, it is important to remember that AI and animation are closely tied together. The AI must be able to correctly select an appropriate animation for playback or the level of realism may be compromised.

C. FUTURE WORK

This thesis has touched on where AI is today, but it is important to look at where Delta3D should go from here, specifically with respect to AI. Part of the motivation of this thesis has been to convey an understanding of why AI is important. Understanding the importance of AI, what are some of the next steps? What would be a reasonable road map to include AI in the Delta3D engine?

1. NPCs Using Python Script

One of the first things that should be attempted is to build a NPC with Python script. Since there is a plethora of web sites dedicated to the modification of Unreal Tournament, this provides a good starting point to begin building the first basic NPC AI using the Python language. Using the Bot class from Unreal Tournament as a template, a similar structure could be developed in the Python language. This structure can become one of the basic AI building blocks in the Delta3D engine and can be a stepping stone to higher AI functions. Using UnrealScript as a model, the functionality that Epic Games has achieved would be an asset to the Delta3D toolset. Already having the graphics rendering, sound and physics, the NPC Python class or classes could spark interest and get the open source community involved. On the Delta3D website, there has been online discussion on developing a game of some type. Of course a game is only one example, but by demonstrating that power and flexibility, further interest would be generated and, hopefully, more developers would join the cause.

2. Interface with Production Systems

Since the DoD is interested in biologically-inspired computing and more closely modeling cognition (DARPA05), the ability to interface Delta3D to an outside inference engine like SOAR or Clips might provide an alternative to entirely writing AI from scratch. However, like many AI technologies, this is context dependent, or put another way, application dependent. One possible downside to this is a developer's lack of familiarity with these specific production

systems and how they work. Similar to Figure 2, the engine interfaces to the production system similar to the gamebots method. The world state information is passed into the production system and is stored in memory which then allows any rules to fire that are satisfied. Once the result is available, it is passed back to the engine.

3. Define an Objective Measure

Potentially, one of the hardest problems is coming up with an objective measure of immersion and intelligence. Specifically, how does one objectively measure the level of "realism" achieved by the AI? This becomes a human factors problem and ultimately requires data collection, and in large numbers to be meaningful or statistically significant.

4. Cost/Benefit Analysis

The issues surrounding cost of licensing the entire engine versus licensing only part of the software (AI Middleware) versus going the open source route were not explored. Since the licensing of the entire Unreal engine is quite expensive, how does it compare to paying a relatively small in house team to continue to research and link together available open source projects to build similar functionality.

While it is certainly easy to look strictly at the cost of licensing versus paying a dedicated team to champion the open source effort here at NPS, it is also important to take a look at the value added by either method. In the case of licensing the Unreal Engine, is the cost worth the benefit of having one of the latest game engines? Or, does the open source approach gain enough momentum that creates a lot of value added by those who contribute their talents and time? Due to the intent of the Delta3D project and what it is trying to achieve, it would appear that the open source approach would be more economical; however, no in-depth study has been conducted.

5. AI on a Dedicated Processor

In the computer graphics field, much of the recent emphasis has been using specialized graphics processing units or GPUs to offload the graphics specific functions and calculations from the Central Processing Unit (CPU). By having dedicated graphics processing hardware, the CPU is free to do other tasks like AI or Physics. One company has prepared a dedicated processor to strictly deal with the idea of physics calculations. This dedicated circuit card removes some of the burden on the CPU with respect to all the motion calculations required for rigid body collisions. Again, the end result should free up the CPU for other tasks as necessary. Currently, the card is not yet available as the computer game industry hasn't fully accepted this idea.

Similarly, it is conceivable that a custom processor card could be dedicated to the calculations and functions specific to the AI in a game and simulation engine. This dedicated AI processor would reduce the load on the CPU and increase the level of AI functionality that could be built into a game or simulation. Though the GPU has been quite successful, at this point it is unclear that a dedicated AI processing unit will receive a similar response.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Abrahams03 Abrahams, “Building Hybrid Systems with Boost.Python.” [http://www.boost-consulting.com/writing/bpl.html]. March 2005.

Bererton04 Bererton, Curt. “State Estimation for Game AI Using Particle Filters” from *Challenges in Game Artificial Intelligence: Papers from the AAAI Workshop*. Technical Report WS-04-04. April 2004.

Cal3d05 Character Animation Library 3D Project Page on Sourceforge. [http://cal3d.sourceforge.net/docs/api/html/cal3dfaq.html]. March 2005.

Champanard05 Champandard, A., “Flexible Animat Embodied aRchitecture (FEAR).” [http://sourceforge.net/projects/fear/]. March 2005.

Darken04 Darken, R. and Johnson, R.E., “Game Engine Comparison Table,” Report Prepared for the Naval Education and Training Command, November 2004.

DARPA05 “Biologically-Inspired Cognitive Architectures (BICA) Proposed Information Pamphlet BAA # 05-18.’ [http://www.darpa.mil/ipto/solicitations/open/05-18_PIP.htm]. March 2005.

Delta3D05 “Delta3D Open Source Gaming and Simulation Engine.” [http://delta3d.org]. March 2005.

Dybsand03 Dybsand, E., “AI Middleware: Getting into Character, Part 1–Part 5.” [http://www.gamasutra.com/features/20030725/dybsand_01.shtml]. March 2005.

Fu02 Fu, D. and Houlette, R., *AI Game Programming Wisdom 2*, “The Ultimate Guide to FSMs in Games,” Charles River Media, 2004.

Gancarz95 Gancarz, M., *The UNIX Philosophy*, Chapter 3, Digital Press, 1995.

Gutschmidt04 Gutschmidt, T., *Game Programming in Python, Lua and Ruby*, Premier Press, 2004.

Isla05 Isla, D., “Handling Complexity in the Halo 2 AI,” Session Notes at the 2005 Game Developers Conference.

Jones03 Jones, T., *AI Application Programming*, Charles River Media, 2003.

Lua05 The Lua Project Page. [http://www.lua.org]. March 2005.

Morgan03 Morgan, D., *Algorithmic Approaches to Finding Cover in Three-Dimensional Virtual Environments*, Master's Thesis, Naval Postgraduate School, Monterey, California, 2003.

OpenAL05 OpenAudio Library Project. [http://www.openal.org]. March 2005.

ODE05 Open Dynamics Engine Project. [http://ode.org]. March 2005.

OSG05 OpenSceneGraph Project. [http://www.openscenegraph.org]. March 2005.

Rabin04 Rabin, S. *AI Game Programming Wisdom 2*, "Common Game Technologies," Charles River Media, 2004.

RBody05 ReplicantBody Project Page [http://www.vrlab.umu.se/research/replicantbody/]. March 2005.

Russell03 Russell, S.J. and Norvig, P., *Artificial Intelligence: A Modern Approach*, 2nd ed., Prentice Hall, 2003.

Scott02 Scott, B., *AI Game Programming Wisdom*, "The Illusion of Intelligence," Charles River Media, 2002.

Simpson02 Simpson, Jake. "Game Engine 101: Part 1" [http://www.extremetech.com/article2/0,1558,244730,00.asp]. March 2005.

Smith05 Smith, P., "A Comparison of Available Artificial Intelligence Options for Open Source Game Projects," EEL6938 Final Project, University of Central Florida, 2005.

Thomas04 Thomas, D., *AI Game Programming Wisdom 2*, "New Paradigms in Artificial Intelligence,". Charles River Media, 2004.

Valdes04 Valdes, R., "In the Mind of the Enemy: The Artificial Intelligence of Halo 2." 17 November 2004. From [http://stuffo.howstuffworks.com/halo2-ai.htm]. February 2005.

Wiki05 "Finite State Machine," from Wikipedia. [http://en.wikipedia.org/wiki/Finite_state_machine]. February 2005.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Christian Darken
   Naval Postgraduate School
   Monterey, California

4. Karl Pfeiffer
   Naval Postgraduate School
   Monterey, California

5. Rudolph Darken
   Naval Postgraduate School
   Monterey, California