Game AI in Delta3D

Christian J. Darken
MOVES Institute/CS Dept.
Naval Postgraduate School
cjdarken at nps dot edu

Bradley G. Anderegg
Alion Science and Technology Corp
banderegg at alionscience dot com

Perry L. McDowell
MOVES Institute
Naval Postgraduate School
mcdowell at nps dot edu

***Abstract*—Delta3D is a GNU-licensed open source game engine with an orientation towards supporting "serious games" such as those with defense and homeland security applications. AI is an important issue for serious games, since there is more pressure to "get the AI right", as opposed to providing an entertaining user experience. We describe several of our near- and longer-term AI projects oriented towards making it easier to build AI-enhanced applications in Delta3D.**

Keywords: Artificial Intelligence, Game Engines

I. Introduction

Delta3D is a open source (Lesser GNU Public License (LPGL)) game engine created at the MOVES Institute, part of the Naval Postgraduate School in Monterey. While originally designed for military game-based simulations, Delta3D can be used as the underlying architecture for a wide range of game applications. Until recently, one difficulty in this was that Delta3D lacked a specific capability to produce artificial intelligence (AI) for either control of non-player characters (NPC's) or for any other aspect a designer would require, e.g., evaluating player performance. We have added such a capability to version 1.4 of the Delta3D engine, greatly expanding the scope of applications which can be built using Delta3D.

We provide an overview of the Delta3D engine in section II, and the basic AI functionality that the engine provides in section III. In sections IV through VII, we give views into more advanced types of AI that we are investigating, including planning, perceptual modeling, learning, and infrastructure to support educational games.

II. The Delta3D Game/Simulation Engine

A. Delta3D Philosophy

In recent years, the military training and operational industries have begun to use solutions from the entertainment industry [1]. Many of the proprietary solutions in the military game and visual simulation industry do not meet the needs of all developers. Licensing costs and restrictions make it difficult to produce the large number of applications required for the myriad of training applications the military requires [2]. Delta3D was created to meet the need for an open source commodity game and simulation engine.

In examining the problems or proprietary solutions for this space, we came up with a four-part philosophical credo upon which we based building our game and simulation engine:

1. Maintain openness in all aspects to avoid lock-ins and increase flexibility.
2. Maintain the capability to support multiple genres, since we never know what type of application it will have to support next.
3. Build the engine in a modular fashion so that we can swap anything out as technologies mature at different rates.
4. Build a community (or leverage existing ones) so the military does not have to pay all the bills.

The first of these, "Maintain openness in all aspects to avoid lock-ins and increase flexibility", addresses two of the problems in the current paradigm. By keeping everything open, no vendor would be able to lock the military into its technology. This would allow any follow-on applications to be bid on by multiple companies, with the resulting competition reducing the costs. Additionally, because the tools are open, developers have access to the source code. This means that if the tools don't meet the developers' requirements, the developers can change the tools as needed for their applications without waiting for a vendor to decide to do so.

The second principle, "Make it multi-genre since we never know what type of application it will to have to support next", is designed to ensure that Delta3D can meet the developer's needs, whatever they are. Within just one of the military services, the Navy, the number of training applications is immense. When he was the commander of the Navy Education and Training Command (NETC) in 2004, Vice Admiral Alfred Harms estimated that he would need approximately 1,500 training games to meet his requirements of performing all individual training (as opposed to team training) within the Navy [3]. There is no single genre of games that will be able to meet all those requirements, which are just a small portion of all those in the military, not to mention the entire spectrum outside the military. While traditionally game engines have been built for a single genre or even a single game, that model would not work for the

military and those interested in a commodity solution. By having one engine which can meet all requirements it is easy to standardize the production pipeline and reuse content for multiple applications, thus reducing the cost involved.

The third of these tenets, "Build the engine in a modular fashion so that we can swap anything out as technologies mature at different rates", will allow the engine to be state of the art for a long period of time. Each of the various elements of the engine consists either of an open source library or code developed in house. In either case, we have kept the different modules as separate as possible. Therefore, if one of the modules making up Delta3D is surpassed by another open source project and is no longer the "best of breed", it is possible to replace that module with the better one. This can continue with only minor modifications to the Delta3D API, thus allowing the engine to remain current

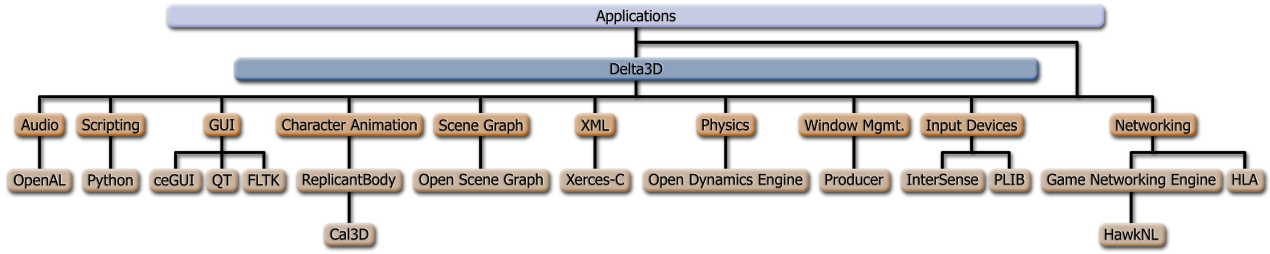

Fig. 1: Delta3D's architecture, showing the external open source libraries and the type of functionality they provide.

significantly longer than most existing game engines.

While longevity was the primary motivation for Delta3D's modularity, it has led to a near-term advantage as well. In some cases, the commodity solution provided by the open source Delta3D module does not meet the needs of the developer. For example, in a close quarters battle simulation, it might be required to use more advanced avatars than those provided by Delta3D and the developer may wish to use a proprietary solution, such as Boston Dynamics'™ DI-Guy™. Because Delta3D is licensed under the LGPL, such proprietary solutions can be integrated without the requirement of being released as open source, such as would be required under the GPL or other viral licenses. This flexibility gives the developer the freedom to choose the best solution in creating an application.

The final part of the credo, "Build a community (or leverage existing ones) so the military does not have to pay all the bills", is another factor driving us towards an open source solution. The power of open source projects is that the energy of a huge development team can be brought to bear upon problems without actually employing such a large team. By building a well designed system that people are interested in using for their own applications, they will also add improvements to the original system. Over time, these may add up to have significantly more value than the original system. However, building such a community takes a great deal of time. Leveraging existing open source communities by incorporating current open source projects with large developer bases into the engine creates a built-in group of developers. The advantages this accrues will be discussed more below.

B.Delta3D Architecture

While building the Delta3D game engine, we had to determine how and what functionality we needed to add to the engine. Our preferred method of adding functionality was using other open source projects which met our requirements and used a license compatible with the LGPL. For example, we used OpenSceneGraph for Delta3D's rendering, Open Dynamics Engine for its physics, etc. Often there were multiple projects meeting these criteria, and in these cases we evaluated the projects for inclusion using two criteria: a project's technical merits and its user support base. The rationale for choosing projects upon their merits is obvious and considering a project's base has allowed Delta3D to gain many "indirect developers", i.e., developers who improve Delta3D by improving one of the underlying open source projects. Additionally, projects with large user bases are more likely to remain current and state of the art than those with only a small base, reducing the likelihood of needing to swap a module. This allowed us to build a robust game engine while minimizing the work required to be done in house.

As far as determining what to add to the engine, we have tried to keep Delta3D extremely lean and have added only those features which are required for the majority of applications. As the use of the engine has expanded, it has become obvious that additional functionality was required, and the need for AI has necessitated adding such a module to Delta3D.

For a more in-depth discussion of the process of creating Delta3D, see McDowell et al. [4]. The initial modules using open source projects, along with the specific projects used for that module, are shown in Fig. 1.

C.Projects Built Using Delta3D

Delta3D has been used as the underlying architecture for several games and simulations, most of them in the serious games arena. Here is a short description of just a few to demonstrate the scope and capability of Delta3D.

1) FOPCSIM: FOPCSIM is a simulation created by USMC students at the MOVES Institute to train forward

observers in the process of controlling artillery. It is extremely militarily accurate, with the player forced to follow exactly the correct military procedures in order to successfully complete the mission. FOPCSIM has been adapted for training by many different agencies in both the US Army and Marine Corps [4]. FOPCSIM lacks certain typical characteristics of games produced for entertainment: there is no background story, the player is unable to move, there is no exploration, etc. Even though FOPCSIM cannot be classified as a true game, it is an example of how game technology can be used for military training outside of traditional games. A screen shot from FOPCSIM can be seen in Fig. 2.

Fig. 2: A screenshot from the FOPCSIM forward observer trainer.

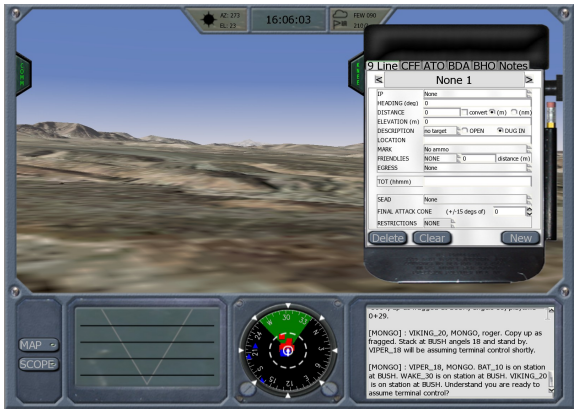

Fig. 3: A screenshot from the Cleared Hot! FACA trainer. The panes at right allow the trainee to communicate with the AI-controlled pilots.

2) Cleared Hot!: Cleared Hot! is a simulation created by USMC students at the MOVES Institute to train FACA's (Forward Air Controllers, Airborne). Like FOPCSIM, it is extremely militarily accurate, with correct adherence to procedures required for successful completion of the tasks. Similarly, it is being adapted for training use in the military [5]. Cleared Hot! has more characteristics of a game than FOPCSIM, such as a background story and the ability to move throughout the environment and explore, nonetheless the design gave very little thought to making the game fun for the player. Therefore, Cleared Hot! cannot be considered a true game. A screen shot of Cleared Hot! is shown in Fig. 3.

3) Driving Trainer: SIMEXAM is a game created by SIMSPACE™, a Spanish company. It is designed to be used by potential drivers preparing for their driving test at the Spanish equivalent of a department of motor vehicles. This is much more of a game than either FOPCSIM or Cleared Hot! The user is allowed to follow the routes expected in the driving test but can also explore the environment. Additionally, the user may "play" by performing actions that would be inadvisable in the real world, such as driving head on into traffic to see the results. A screen shot from SIMEXAM is shown in Fig. 4.

III. Basic AI Support

Delta3D offers three basic facilities to support the development of AI, namely a finite state machine class, traditional waypoint-based navigation, and the ability to code AI in a high-level scripting language. All three types of functionality have been widely used in the video game industry, so there are multiple example implementations to draw from. Furthermore, the usefulness of this functionality is widely acknowledged. Of the three items, finite state machines are perhaps the most universally adopted.

Fig. 4: A screenshot from the SIMEXAM driver training game.

Finite State Machines (FSM's) are one of the principle models used in theoretical computer science. They have also been widely adopted for use in game AI. FSM's are composed of a finite set of states and a set of transitions amongst them. There is also a finite set of input symbols ("events" in Delta3D) that cause the state to change. Each transition is labeled with an input symbol. At any moment, the AI is said to be "in" one of the states. When a new input symbol is sent to the FSM for processing, the set of transitions from this state to any other is checked to see if any of them is labeled with the new symbol. If one transition is, the current state is changed to be the one indicated by the transition. Implementing a FSM is no great challenge for an experienced programmer, but FSM support belongs in a game engine because the small amount of time required for implementation multiplied by the number of projects that

need the functionality (likely to be a large number) would be considerable.

Some sort of navigation infrastructure is necessary to allow AI to move about a game level. Navigation infrastructure does for the AI what the player's eyes do for the player: allow it to move around the level without colliding with walls and other obstacles. The navigation infrastructure implemented in Delta3D is based on waypoints, i.e. mathematical points in a level that can be used as intermediate destinations within a longer path. The waypoints constitute the nodes of a navigation graph. For each waypoint, a directed edge is added to the graph for each other waypoint that can be moved to along a straight line. Navigation infrastructure generally has an off-line component that is used when a level is built in order to construct the navigation graph, and an on-line component used at run-time to actually plan paths.

Delta3D's level editor, STAGE, has been enhanced to allow the user to add waypoints. This is accomplished via a manual graphical editing process. Waypoints are visualized as simple billboards, and the user locates them in the level just as he would locate an object such as a tree or rock. Upon saving the level, a ray is traced between nearby pairs of waypoints, and if the ray does not intersect anything, a directed edge is added to the navigation graph. This is an inexpensive substitute for a more exact test of the ability to move between the waypoints in a straight line that is accurate enough, if a little care is exercised in waypoint placement.

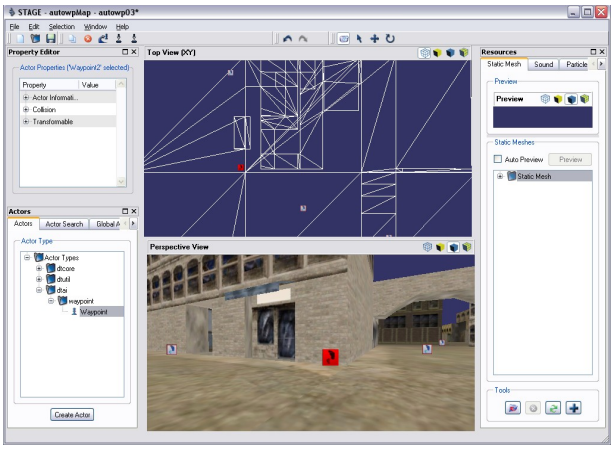

Fig. 5: Manual waypoint placement in the STAGE level editor. The currently manipulated waypoint is red, and the others are gray.

At runtime, the API provides a function that will attempt to plan a route between any two points (not just waypoints) in the level. This is accomplished by searching the navigation graph using the A* algorithm. The Delta3D A* implementation is resumable, so the computation of difficult paths can be split over multiple frames. The function returns the shortest path to the goal location in the form of a list of waypoints. Actual use of the waypoints to move an agent is done by user code. The waypoints can be traversed in a simple "connect the dots" manner, or by skipping earlier waypoints on the path in favor of the latest visible one, a trick known as "string pulling" [13], which often looks more natural, though its success in arbitrary level geometry cannot be taken for granted. Note that run-time collision checks with the static geometry (walls, etc.) of the level can be skipped while on the navigation graph, as movement is only between pairs of waypoints determined to be traversable when the level was built.

Delta3D allows the user to write partial or complete programs in the high-level scripting language Python. Not every game engine provides a scripting language. The arguments for doing so are that scripting languages are typically easier to learn and faster to use, at least for smaller programs. This translates into a lower bar for contributing to the code base, allowing people with a much broader range of backgrounds to modify a simulation in fundamental ways. Less time is often required to produce a given behavior. And since scripting languages tend to be dynamically typed, code can often be re-targeted more quickly, since the overhead of dealing with types is avoided.

The Python language has been widely adopted for a wide range of uses, one of which is as a scripting language for games and simulations. Python is very widely used, and has a large and useful standard library. It has a streamlined Java/C-like syntax that is very concise, making codes small, and bugs easier to locate. It has excellent list and associative map (dictionary) support, which are of particular importance for developing AI. It features "just-in-time" compilation, so there is no explicit compile step, resulting in a fast modify-and-test cycle.

Delta3D provides Python bindings corresponding to many, but not all, of the calls in its C++ API. Boost Python is the tool used to add new bindings. Python bindings for some functions are not easy to produce. Conceptual effort may be required, since there is a considerable gap between the data types and memory management of C++ and Python, and Boost Python is a very sophisticated tool whose behavior is not always easy to understand and manipulate. In the vast majority of cases, however, Boost Python makes producing the bindings quick and straightforward.

IV. Planning

Most games and simulations produced to date do not use any technology from the field of AI as the top-level architecture for the AI in favor of some flavor of Finite State Machines or a custom software design. In 2005, Monolith's game F.E.A.R. became the first AAA title to use a streamlined version of AI planning for this purpose [12]. Jeff Orkin, one of the developers of F.E.A.R.'s AI, found that Finite State Machines did not scale well to large AI's [11]. Delta3D is perhaps the first game engine to offer a form of planning built in to the engine.

Delta3D's planning facility is based on forward state-based search conducted by the A* algorithm. We chose forward state-based search because it is well suited to

domains with limited continuous-valued resources such as fuel or ammunition. The planner's API requires the user to represent all actions that are possible to the agent in terms of Operator objects. Operators describe the necessary preconditions to take the action as well as its effects, and any events that would necessitate re-planning ("interrupts").

Delta3D's planning capability was used for the first time in the *Cleared Hot!* trainer, where it was used to represent the AI fixed-wing and helicopter pilots. As in Orkin [11], we used a small Finite State Machine to bridge between the plan and the rest of the game engine. We found the planning infrastructure very well suited to representing the complex, multistage, doctrinally prescribed interactions amongst the pilots and the trainee.

Delta3D's C++ planning API is expected to be released soon. In recent work, we have prototyped the capability to describe the planning problem in a custom scripting language. An example appears as Fig. 6. User code is used to make the operators specified in the script affect the state of the game, e.g. to actually move the agent around. An example planning script for an agent that continuously hides whenever he becomes visible is provided below. There are multiple goals that can be switched among by user code at run time. The numbers attached to the goals are estimated total cost to achieve the goal. The "Idle" goal is the default, and results in the agent doing nothing and waiting for the "IsHidden" goal to be made active.

Hider Planning Script

```
NPC Hider
[

WorldState [Idle(true),
      HideFailed(false),
      HavePlaceToHide(false),
      HavePathToWaypoint(false),
      IsHidden(false)]

operator FindPlaceToHide
   preconds []
   interrupts []
   effects [ HavePlaceToHide(true) ]
   cost 1

operator FindPath
   preconds [HavePlaceToHide(true)]
   interrupts []
   effects [ HavePathToWaypoint(true) ]
   cost 1

operator Hide
  preconds [HavePlaceToHide(true),
      HavePathToWaypoint(true),
      HideFailed(false)]
   interrupts [ HideFailed(true) ]
   effects [ IsHidden(true) ]
   cost 1

operator ReHide
   preconds[ HideFailed(true) ]
   interrupts []
   effects [ HideFailed(false) ]
   cost 1

Goals [ Idle(0, true), IsHidden(3, true) ]

]
```

Fig. 6: A planning script for an AI-controlled character who continuously hides.

V. Perceptual Modeling

For games with a military flavor, the traditional approach to determining battlefield visibility is lacking, with severe consequences for the AI [14]. Most video games use a simple Line-of-Sight (LOS) trace from the eye of the observing agent to the top of the target to make a visibility determination. If the LOS does not intersect any polygons, the agent is considered to be fully aware of the target. Of course, this is a drastic oversimplification, and there are many instances in which LOS visibility is far too generous or too conservative. The result is AI that sometimes engages targets that are invisible to a human player and sometimes fails to engage obvious targets. We are exploring alternative methods of modeling visibility based on a military-developed algorithm called ACQUIRE [16]. ACQUIRE was developed in order to model night vision equipment, so it is not the perfect tool for modeling target detection by the unaided human eye. Nonetheless, there is a long history of using ACQUIRE for this purpose in military modeling and simulation. The advantage of ACQUIRE over LOS is that ACQUIRE considers the amount of exposed surface area of the target and its contrast with its background. However, ACQUIRE has never been previously applied to video game-like simulations.

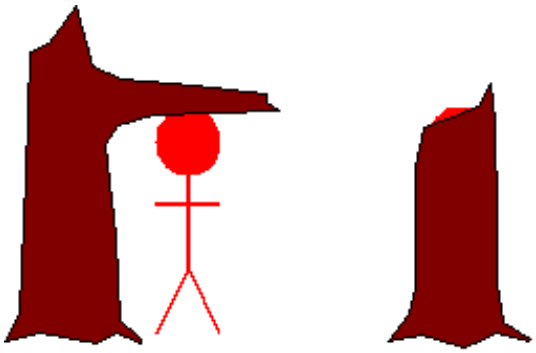

Fig. 7: An illustration of some of the problems with using traditional Lines of Sight for target detection. The figure at left is invisible according to LOS, since the line is drawn to the top of the target, which is obscured behind a tree limb. By contrast, the figure at right, completely obscured except for the top of the head, is visible according to LOS.

We have produced and tested a version of ACQUIRE called GBBA (Graphics Buffer Based ACQUIRE) in Delta3D [15]. GBBA performs two tight "mini-renders" of the target from the agent's point of view: one in normal color and other with the target in false color. The false color render is used to segment the mini-render into target and background. The depth buffer is then accessed in order to remove "background" pixels that are actually in front of the

target. The resulting sets of foreground and background pixels are then used to determine visible target surface area and contrast. Area and contrast numbers then feed ACQUIRE, which then produces detection probabilities. While the superiority of GBBA to LOS as a model of target detection is unquestionable, we have found many fidelity issues with GBBA stemming mostly from factors that are not considered in ACQUIRE such as color, shape, and texture. We are continuing our efforts to enhance GBBA to solve these problems.

Fig. 8: A wider shot of a figure at a window (left), a minirender of the same figure (middle), and a false color minirender with the target in red and the background in green, except for the part that is closer to the camera than the target (the edge of the window), which is colored blue (right).

VI. Learning

Game AI that is self-programming in the sense that it can autonomously learn from experience has been a continuing interest of many in the game industry. Many approaches, such as neural networks, genetic algorithms, and so forth have been explored. To date, learning has not established itself as a standard tool for game AI. Some possible reasons for this failure include the difficulty of understanding and coding the learning algorithms, loss of control perceived by game developers turning over aspects of the AI to autonomous software, difficulty in testing adaptive AI due to the broader range of possible behavior and the difficulty of reproducing problems, and just general poor performance for the intended application. Some of these problems might be ameliorated by building learning into the game engine level building tool set, and by focusing on learning those parts of the AI that seem to have the greatest payoff and the least risk. This is the approach we are taking with Delta3D. Most of our learning work currently focuses on the task of learning to navigate around a level.

As described above, Delta3D's waypoint-based navigation system requires a level designer to place waypoints on the level manually. We are working on removing this requirement by generating the set of waypoints automatically. Our approach seems to be similar to that taken by multiple game industry groups, though our approach was conceived of and developed completely independently, and the algorithms used by others have not been published as far as we are aware. In our approach, the level designer places at least one single waypoint on the level. Multiple waypoints may be placed if desired. An off-line exploration process then takes place where the AI, driving one of the same motion models available for human users, explores the level and learns where it is possible to go in the level and how to get there. At the end of the run, this information is then saved off as a waypoint graph in the same format as generated by a human level designer constructing a waypoint graph manually in STAGE, the Delta3D level editor.

Level Exploration Pseudocode

```
For each waypoint
  For each direction that is a multiple of 60°
    Attempt to move the waypoint spacing
      distance out from the waypoint
    Monitor progress and fail if too slow
    If close to the goal and not falling or
      sliding, add a waypoint at goal (if not
      already there) and add the appropriate
      edge to the waypoint graph
```

Fig. 9: Pseudocode for the level exploration algorithm.

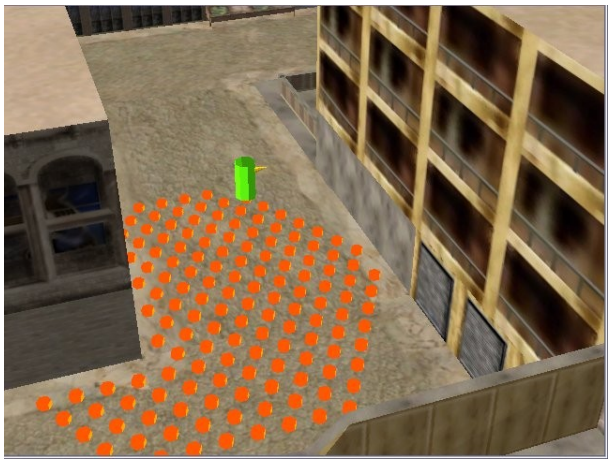

Fig. 10: The level explorer (green cylinder roughly at center, with its facing indicated by the yellow "beak") in the early stages of exploring a level, dropping orange waypoints as it goes.

Fig. 11: The same level being navigated by a figure at run-time. The waypoints have been made visible for illustrative purposes. Most are blue, except for the green ones, which are part of the current path, and the red one, which is the goal.

We plan to enhance the explorer with the ability to evaluate waypoints found as firing or hiding positions. A fine grid of waypoints fixed at level build time, while still prone to some types of problem we have previously described [6] [10], would be highly useful for near-term simulations and games. Note that this grid is built finer than needed for merely moving about the level in order to support behaviors like moving into cover. With a fine grid, the covered positions can be taken to be a subset of the waypoints.

At a more fundamental level, we are investigating the application of general predictive models in the context of games and simulations. The idea is to apply the predictive models to produce agents that are more proactive than current generation agents. These models require a generic representation of game events. Given a stream of such events, they learn to predict what events are likely to be next. We have prototyped this idea using a text-based MUD game [8] [7] as well as a next-generation analytical military simulation [9]. We see no barrier to its application to 3D games of all sorts.

VII. Other Applications of AI

The traditional place where AI fits into games is in controlling NPC's, building behavior into characters so that they appear human (if human) or give the player a realistic challenge to overcome (whether human or not). We will be using the AI added to Delta3D to add this behavior to games and simulations created by Delta3D, but we also see this AI being used to add capabilities to make applications work much better in a serious game environment.

Currently, most instances of using games for learning involve using the game as a separate entity or, at best, efforts where games were not fully integrated into the non-game portions of a class. Additionally, other than fairly simple games where the game can adequately assess the player, the behavior of the student in a game environment must be evaluated by an instructor. This prevents the game from being fully utilized for the training required.

A. Integration into Coursework

In order to fully integrate games into a class, especially one being taught via distant learning, the game must be able to work seamlessly with a learning management system (LMS). LMS's are essentially databases which contain all the information required for a class, such as syllabi, reading lists and material, written assignments, and generally run in a browser window. Quite often, the LMS is used to give on-line quizzes and tests. Additionally, they often serve as the teacher's gradebook, where all the students' scores are recorded and tabulated.

To make a game which is fully integrated with the LMS, the game must be able to be launched from inside the LMS. That is, the student logs into his LMS to find his class assignment; on this day, it is to play a game. The student clicks on a link which starts a game on the student's computer. Once complete, the student's performance should be sent to the LMS, which then adds the performance to the gradebook. Delta3D has recently added the capability in its 1.4 version.

However, there are still significant improvements which can be made to improve the performance of games in educating students. For example, the scenario in the game could be tailored to the student, that is, the LMS provides the game a list of the capabilities of the student and the game automatically creates a scenario based upon these. In the course of the play, the game will adjust its level of difficulty; if the student is struggling, the game becomes easier, while if the student is not being challenged the game can increase it difficulty. At the end of the game, the game determines the objectives the student has demonstrated mastery and reports this to the LMS. These technologies will require a significant AI capability. We are planning to use the AI capabilities added to Delta3D to investigate these areas.

B. Automated Tutoring Support

Another improvement required to help games meet their potential as educational tools is to allow the games to tutor the students. One of the biggest worries is that students will take away the wrong lesson from a training simulation, which is referred to as negative training. As Dr. Jeff Wilkinson, program manager for the Institute of Creative Technologies (ICT), said at the 2006 Serious Game Summit, "Experiential learning is not effective; guided experiential learning is effective." Without some sort of feedback, it is easy for the user to learn behavior which could result in problems in the real world. Unfortunately, except for very simple tasks, games don't currently have that capability. This is mainly because evaluating students' behavior requires a great deal of intelligence to fully understand whether the objective was met and, if not, what the cause of the student's failure was. Additional AI capabilities make it possible to begin investigations into how to evaluate student's behavior and tutor the student.

VIII. Conclusions

Where is the best place to draw the line between the AI functionality provided by a game engine, and that which the user must provide on his own? We have described the Delta3D view of basic AI support, and we expect that line to move a lot as the entertainment and serious game communities continue to gain insight into which AI techniques are the most robust and widely applicable.

References

[1] M. V. Capps, P. L. McDowell, & M. Zyda, A future for entertainment-defense research collaboration. IEEE Computer Graphics and Applications 21(1), 37-43.

[2] R. P. Darken and P. L. McDowell, "Open Source Game Engines: Disruptive Technologies in Training and Education", Proceedings of the 2005 Interservice/Industry Training, Simulation and Education Conference (I/ITSEC). Orlando, Florida: National Defense Industrial Association. November 28 – December 1 2005

[3] A. G. Harms Jr., “Investing in people”, presentation given April 6, 2004, Pensacola Fl.

[4] P. L. McDowell, R. P. Darken., R. E. Johnson, & J. A. Sullivan, Delta3D: a complete open source game and simulation engine for building military training systems. Proceedings of the 2005 Interservice/Industry Training, Simulation and Education Conference (I/ITSEC). Orlando, Florida: National Defense Industrial Association. November 28 – December 1 2005.

[5] D. Kunde and C. Darken, “A Mental Simulation-Based Decision-Making Architecture Applied to Ground Combat”, *Proceedings of BRIMS 2006.*

[6] C. Darken and G. Paull, “Finding Cover in Dynamic Environments”, *Game AI Programming Wisdom 3*, Charles River, S. Rabin editor, 2006.

[7] C. Darken, "Heuristic Speed-Ups for Learning in Complex Stochastic Environments", *Proceedings of IJCAI 2005 Workshop on Planning and Learning in A Priori Unknown or Uncertain Domains*.

[8] C. Darken, "Towards Learned Anticipation in Complex Stochastic Environments", *Proceedings of AIIDE 2005*.

[9] D. Kunde and C. Darken, "Event Prediction for Modeling Mental Simulation in Naturalistic Decision Making", *Proceedings of BRIMS 2005*.

[10] G. Paull and C. Darken, "Integrated On- and Off-Line Cover Finding and Exploitation", *Proceedings of GAME-ON 2004*.

[11] J. Orkin, “3 States & a Plan: The AI of F.E.A.R.”, *Proceedings of the Game Developer's Conference 2006*.

[12] J. Orkin, “Agent Architecture Considerations for Real-Time Planning in Games”, *Proceedings of AIIDE 2005*.

[13] P. Tozour, “Search Space Representations”, *AI Game Programming Wisdom 2,* Charles River, 2004.

[14] C. Darken, “Visibility and Concealment Algorithms for 3D Simulations”, *Proceedings of BRIMS 2004.*

[15] B. Jones, “A Computer Graphics-Based Target Detection Model”, Master's dissertation, MOVES Institute, Naval Postgraduate School, Monterey, CA 2006.

[16] D. Reece and R. Wirthlin, “Detection Models for Computer Generated Individual Combatants”, P*roceedings of the 6th Conference on Computer Generated Forces and Behavioral Representation*, 1996.