The Design and Implementation of a Testbed for Comparative Game AI Studies

Hollie Boudreaux, Jim Etheredge, Ashok Kumar

Ashok Kumar: axk1769@louisiana.edu
Hollie Boudreaux: hollieboudreaux@gmail.com

***Abstract*—An essential component of realism in video games is the behavior exhibited by the non-player character (NPC) agents in the game. Most development efforts employ a single artificial intelligence (AI) method to determine NPC agent behavior during gameplay. This paper describes an NPC AI testbed under development which will allow for a variety of AI methods to be compared under simulated gameplay conditions. Two squads of NPC agents are pitted against each other in a game scenario. Multiple games using the starting same AI assignments will form an epoch. The testbed allows for the testing of a variety of AI methods in three dimensions. Individual agents can be assigned different AI methods. Individual agents can use different AI methods at different times during the game. And finally, the AI used by one type of agent can be made to differ from the AI used by another agent type. Extensive data is collected for all agent actions in all games played in an epoch. This data will form the basis of the comparative analysis.**

***Index Terms*—Artificial intelligence, testbed, video game studies, gameplay simulation**

I. Introduction

THIS paper discusses the implementation of a testbed to be used as a mechanism for research into comparative artificial intelligence techniques as applied to NPCs in video games. Most video games use a single AI methodology, such as pathfinding, rule-based systems, or neural networks. The intent of the testbed is to allow the AI technique to be varied during gameplay.

The approach is to create a generic game environment that allows complete flexibility in the number and composition of the agent squads. There are several agent types with inherent specialties and personal goals. Various AI techniques are being implemented and can be assigned in real time. Game control is accomplished via the use of a user interface which offers detailed control over the environment, squads, and agent parameters. The testbed also includes extensive event logging for immediate feedback as well as post-trial in-depth comparative analysis of AI methods.

II. Related Work

Aha and Molineaux [1], [2] propose a testbed named Testbed for Integrating and Evaluating Learning Techniques (TIELT). The goal of TIELT is to ease the integration of artificial intelligence systems with gaming simulators, but the current status of TIELT's development is not known.

Coiana et al. [3] present a testbed platform for the iterative design of multimodal games on a mobile phone or a PDA. The work focuses on games on mobile devices. The work takes a player centered design approach and aims to quickly test various forms of multimodal interaction. The testbed is reported to facilitate efficient exploration of the multimodal design space.

Work by Dawest and Hall [4] highlights that embedding a cognitive model into stable artificial intelligence characters is non-trivial, and proposes an intermediate architecture for a first person shooter game and artificial intelligence. The feasibility of the architecture is reported to have been assessed within the game Unreal Tournament 2004.

Fullam et al. [5] introduce a testbed named Agent Reputation and Trust (ART). The ART testbed initiative is charged with the task of establishing a testbed for agent trust and reputation related technologies. This testbed serves two roles: (1) as a competition forum in which researchers can compare their technologies against objective metrics, and (2) as a suite of tools with flexible parameters, allowing researchers to perform customizable, easily repeatable experiments.

Testbeds reported in the literature do not provide for multiple types of agents with each potentially using a different AI method, whereas the proposed solution does provide these features. The eventual goal of the proposed solution is to compare the effects of many AI methods among members of an agent squad to determine which combination provides the most effective team. In addition, the effects of changing the AI of an agent mid-trial will also be examined.

III. Testbed Design and Implementation

The testbed is implemented in C++ using the Object-Oriented Graphics Engine (Ogre) [6]. Ogre is free, open source, and supported by a large online community. The interface was created using CEGUI [7], an external library for Ogre. The testbed terrain was created using EarthSculptor [8] and the agent models were created using Maya [9].

A. User Interface

The main window contains buttons for accessing the other windows. For example, pressing the "Agent Parameters" button will launch the corresponding window. A drop down box is available along the top of this window that contains all available agents. When an agent is selected all the parameters associated with that agent are displayed and can be edited if desired.

The game creation window, shown in Fig. 1, allows the user to specify the number of agents of each type for each squad. The testbed then creates the desired number of agents with

Manuscript received June 30, 2010.
The authors are with the University of Louisiana at Lafayette.

DOI: 10.5176_2010-2283_1.1.01

Fig. 1: The Create Game Window

default parameters and places them into the game world. The placement is selected by the user to either be random or around a specified squad base location. A check is performed to ensure an agent is not placed inside a wall or other obstacle. Adding a single agent is also possible through a different window and is used to create a new agent with full control over its parameters. All available parameters have text boxes where the user can specify the desired value or leave it blank to use the default value. The user can choose to have the agent placed into the scene at a location specified as the squad base or at the location of a mouse click.

After all agents have been placed, the user can access the buttons used to control a trial via the "Play Game" button on the main menu. This brings up another menu with options to start, stop, pause, step, or reset the current trial. Fig. 2 shows this menu in addition to a group of agents about to begin a trial. The start and stop options will begin and end a trial, respectively. Pause will temporarily halt a trial in progress or resume one that is already paused. Fig. 3 shows a paused trial with several defeated agents. Step allows the user to run one game cycle at a time. Reset will return all agents to their initial position and set all parameters to their original configuration. As can be seen in Fig. 2 and 3, the agents' representation is purposefully unadorned to avoid negatively impacting testbed performance.

IV. Detecting Other Agents

The formula for the dot product of two vectors provides a simple method for determining if an agent is within the viewing range of another. The algorithm used is shown in Fig. 4.

For example, assume agent A has a viewing angle of 60° and is trying to determine if he can see Agent B. If the dot product between the two normalized vectors is less than the cosine of 60°, then A can detect B, provided that the distance between A and B is less than A's maximum viewing distance. Detected agents are added to either the enemy agents seen or

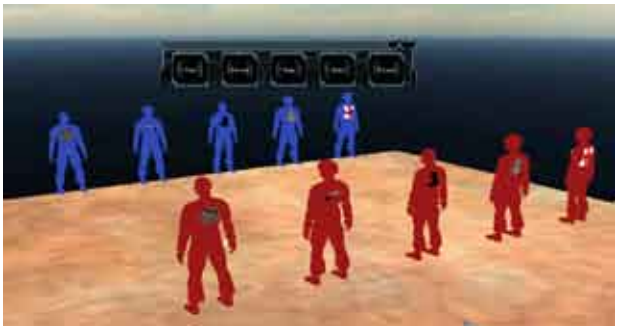

Fig. 2: Pre-Trial Setup

Fig. 3: A Trial in Progress

ally agents seen lists as appropriate. Next, the communication range is used to determine ally agents with which each agent can currently exchange information. Ally agents exchange information by exchanging lists of visible enemy agents and visible ally agents. As a result, an agent can obtain much more information about the world than would normally be available. In a future phase of the testbed, an option to display each agent's viewing area during gameplay will be added.

```
vector3 lookDirection = current orientation (Quaternion)
* agent's default direction (vector)
for all other agents in the environment do
  vector3 targetDirection = target's location - agent's
  location
  if length of targetDirection ≤ agent's maximum view-
  ing distance then
    normalize lookDirection and targetDirection
    if dotProduct of lookDirection and targetDirection
    > cos(agent sight angle) then
      if target's allegiance = agent's allegiance then
        add target to list of ally agents seen
      else
        add target to list of enemy agents seen
      end if
    end if
  end if
end for
```

Fig. 4: Algorithm for the Detection of Agents

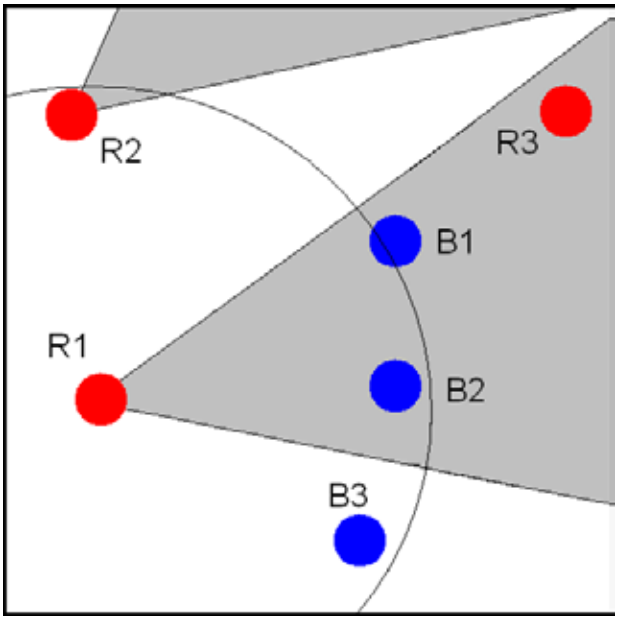

Fig. 5: Simplified Game Environment

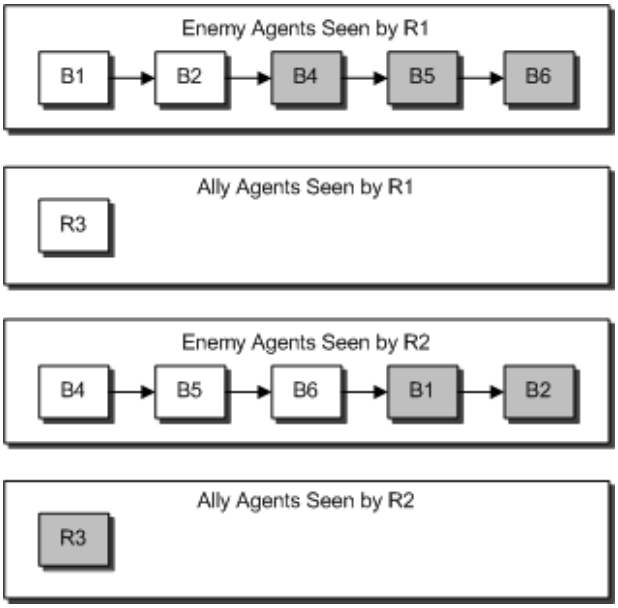

Fig. 6: Agents Seen Lists for R1 and R2

A simplified representation of a portion of the game environment is shown in Fig. 5. R1 and R2's viewing areas are shaded in gray. R1 can see agents B1, B2, and R3, while B3 is just out of range. Assume that R2 can see agents B4, B5, and B6, which are not shown in Fig. 5. The communication range of R1 is represented by the circle surrounding it. Since agent R2 is within this range, R1 and R2 can share information. The list of visible agents for both R1 and R2 is shown in Fig. 6. Shaded boxes represent information that is gained via this communication process.

When information is exchanged between agents R1 and R2, R1 gains knowledge of the locations of B4, B5, and B6 and appends their names to the end of its list of visible enemy agents. Similarly, R2 learns the positions of B1 and B2 from R1. When R1 and R2 share information about ally agents R2 gains knowledge of R3 from R1. Agent R3 cannot share information since it is not within communication range of either R1 or R2.

V. Preserving Agent Configurations and Parameters Between Trials

A user may wish to run multiple trials with identical agent starting positions and different parameters. A record of all actions taken during a trial would also be useful. Several utilities and extensions to the testbed have been created to accomplish these goals.

A. Logs

The logging utility was created in order to keep a record of all actions performed during a game or trial. Every possible action that can occur, such as creation, deletion, movement, and attacking, has an associated log entry type. When an action is performed, a log entry is written both to the screen and to a log file. The log file entries only save the minimum amount of data, as opposed to the screen entries which are more verbose. For example, the screen entry "RedSoldier1 attacked BlueHealer2 for 5 damage" would be written to the file as "7 RedSoldier1 BlueHealer2 5." In the preceding example log entry the 7 represents the action attack within an enumerated list. In a future version of the testbed the log will be used to "save" a trial or game in progress and reconstruct the scenario at a later time in order to continue from where it was suspended.

A separate log parser program has also been created. It can read the log files and display the entries in a human readable format. The parser also allows for the filtering of log data by one or more entry types and one or more agents. For example, this allows a user to check for all attack actions performed by a specific agent.

B. Level Editor

While the ability to place agents exists on the testbed, it will eventually be expanded to contain obstacles such as walls and objects that can be used for cover. It was decided that a separate program to set the initial position of agents and environmental objects would be useful to avoid loading the testbed down with features unnecessary for performing its primary purpose. The Level Editor, shown in Fig. 7, was created to serve that purpose.

The menu shown on the left side contains a radio button for each type of object and agent available. The object whose radio button is clicked will be placed on the terrain at the position of a left mouse click. There is no restriction on the number of objects that can be placed this way. An object can be moved using a click and drag operation. Selecting groups of agents to move simultaneously is also supported. After creating the desired layout, clicking the save button on the right side menu will create a .lvl file specifying all objects present in the scene, their locations, and their orientations. This file can then be loaded into the testbed application. When

Fig. 7: The Level Editor

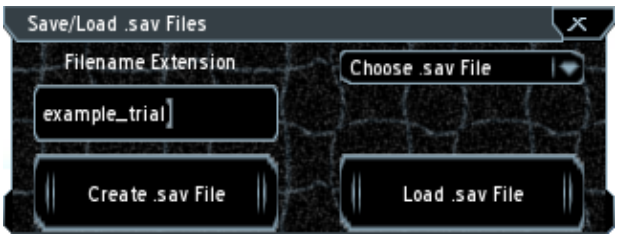

Fig. 8: The .sav File Menu

using this method, agents are created with default parameters, but these can be altered at the user's discretion via the Agent Parameters window.

C. .sav Files

A user may wish to run multiple trials with the same configuration of agents but varying parameters. The .lvl files do not include any parameter information aside from type, location, and orientation. The .sav files were created in order to preserve changes made to agents parameters in the Agent Parameter window. A user can create a standard layout of agents and environmental objects using the Level Editor and use the resulting .lvl file as a basis for creating many .sav files with different parameter configurations. When the file is loaded, all existing agents are deleted and new agents are created using the parameter information specified in the .sav file.

Pressing F12 or the .sav file button on the main menu will launch the .sav file window. On the right side, as shown in Fig. 8, a string can be specified. This string will be appended to the filename, allowing the user to easily distinguish between multiple .sav files. The user can select a .sav file from the dropdown list on the left. When the load button is pressed, all existing agents are deleted and new agents are created using the parameters specified in the .sav file.

VI. Agents

This section will discuss the differences between agent types, agent parameters, and the representation of an agent.

A. Agent Types

Five types of agents are used in the testbed. The Soldier is a generic agent that can use a weapon. The Tank agent is more powerful than a Soldier, but moves considerably slower. Scouts can see more of the environment and move faster than any other type, but are relatively weak. Healer agents increase the health of wounded ally agents, and Suppliers provide resources to ally agents. Both the Healer and Supplier are unable to attack other agents.

B. Agent Parameters

Many of the agent parameters used in the testbed fall into one of the following categories:

- World Representation This includes position, orientation, and size parameters.
- Detection The viewing range and distance (collectively called fieldOfView) and communication distance fall into this category.
- Agent interaction Separate lists are used to keep track of enemy and ally agents that can be detected by a particular agent. An additional list is used to monitor ally agents that are in communication range.
- Resources This category covers items needed for an agent to function, such as ammunition or bandages and health.

Other parameters used include expertise level, squad allegiance, AI method used, and maximum movement speed. New parameters will be added as the need arises.

C. Agent Actions

An agent may perform one action per time step. This agent can choose to perform either a move or resource consuming action. Actions such as fleeing from an enemy, patrolling an area, or flanking a target are all examples of movement actions. Firing a weapon, restoring health, or providing supplies are resource consuming actions.

D. AI Interface

On each update cycle, each NPC's action is determined separately using a combination of its assigned AI methodology and a randomization factor to add a nondeterministic element to the game. Fig. 9 shows the decision process for updating an agent's AI. The agent's representative bit string is passed to the AI Interface layer and based on the value found in the AI field it is sent to the appropriate AI module. As a result of this representation, the numbers and types of AI represented are independent of the rest of the program and each agent does not have to use the same AI. During each update cycle, each agent's parameter bit string is submitted to the AI interface. The AI interface uses the agent's assigned AI module to determine appropriate actions. The testbed will complete each agent's update by applying the selected action. For example, the AI interface may suggest multiple possible action for an agent leaving the testbed free to generate random movement based on probability. This approach will allow the testbed to include an element of randomness into the gameplay without subverting decisions made by the AI interface.

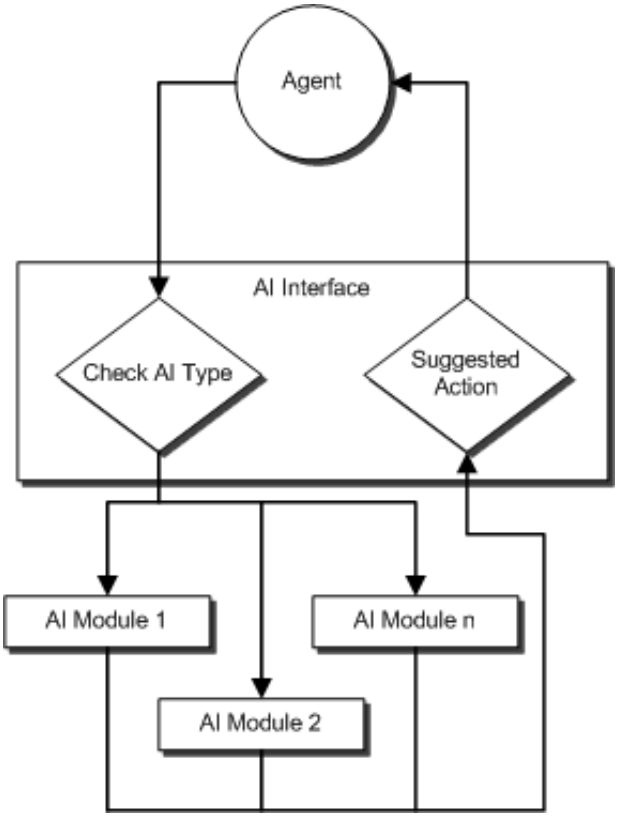

Fig. 9: AI Interface Architecture

VII. Test Epochs

In order to provide adequate test data it is necessary to run a game multiple times using the same parameters. This is accomplished in the testbed through the use of epochs. An epoch is the same game run multiple times using the same starting parameters. The starting parameters include:

- The number of games in the epoch.
- The number of agents on each squad.
- The type of each agent.
- The starting position of each agent.
- The resources associated with each agent.

Once the basic game environment has been specified by the user, the testbed will run the epoch by starting a game and letting it run to its conclusion. When the current game is over, the testbed will reset the game environment to the original configuration and begin the next game in the epoch. When the specified number of games have been played, the epoch will be terminated. Data logged for the epoch will be an aggregate of the data logged for each individual game played.

There must be some form of randomness in the gameplay so that running the same game over and over does not produce the same results each time. This is accomplished by the inclusion of probability factors associated with various events that occur in the course of gameplay. The following list describes some of the basic events and game parameters where the introduction of randomness can have a significant effect on the final outcome of the game.

- When no movement direction is indicated by the AI, an agent can choose to move in a random direction.
- When an agent attacks, the success of the attack can be determined in part by the agent's expertise level.
- When an agent is wounded, its chances of survival can be partially determined by a probability factor.
- When faced with multiple possible actions, an agent can choose one at random.

One example of this type of randomness can be seen in the agent expertise parameter. This parameter can be used to add an element of randomness to the agents behavior. The lower the expertise level of an agent, the more likely they will be to make random decisions that result in actions not dictated by the AI.

The overall effect of introducing randomness into the gameplay is to produce enough variability to ensure that no two games will be exactly alike. However, caution must be exercised to also ensure that the randomness does not invalidate the behavior guidance provided by the AI. Toward this end, the probabilities used to create randomness will generally be set relatively low.

VIII. Conclusion

The testbed described in this paper is a tool to be used to facilitate research into the comparative strengths and weaknesses of various artificial intelligence methodologies used to guide the actions of non-player agents in video games. As such, it is designed to provide a general purpose game environment while, at the same time, creating a realistic gaming look and feel. Its primary contribution to the research is the collection of data that can be analyzed to determine the effect of AI methodologies on the behavior of NPCs in a game environment. The development of the testbed is planned for several phases. After the first phase, which is the initial creation of the basic testbed, subsequent phases will enhance the data collection capability and add features to the game environment. The ultimate goal of the project is the collection and analysis of data gathered during gameplay involving multiple AI methods used in tandem at both the agent and squad level. Subsequent phases of development will focus on fine tuning the testbed's data collection capability and adding additional features to extend the complexity of the game environment.

Acknowledgment

The authors would like to acknowledge Devin Faulk, Joshua Hebert, Jennifer Lavergne, Phillip Spear, and Devin Rooney for their assistance with the implementation of Phase I of the testbed.

References

[1] D. W. Aha and M. Molineaux, "Integrating learning in interactive gaming simulators," in *Challenges in Game AI: Papers of the AAAI04 Workshop (Technical Report WS-04-04)*. AAAI Press, 2004.

[2] M. Molineaux and D. W. Aha, "Tielt: a testbed for gaming environments," in *AAAI'05: Proceedings of the 20th national conference on Artificial intelligence*. AAAI Press, 2005, pp. 1690–1691.

[3] M. Coiana, A. Conconi, L. Nigay, and M. Ortega, "Test-bed for multimodal games on mobile devices," in *Proceedings of the 2nd International Conference on Fun and Games*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 75–87.

[4] M. Dawes and R. Hall, "Towards using first-person shooter computer games as an artificial intelligence testbed," in *Knowledge-Based Intelligent Information and Engineering Systems*, 2005.
[5] K. K. Fullam, T. B. Klos, G. Muller, J. Sabater, A. Schlosser, Z. Topol, K. S. Barber, J. S. Rosenschein, L. Vercouter, and M. Voss, "A specification of the agent reputation and trust (art) testbed: experimentation and competition for trust in agent societies," in *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2005, pp. 512–518.
[6] "Ogre Website," http://ogre3d.org/, 2009.
[7] P. Turner, "CEGUI," http://www.cegui.org.uk/wiki/index.php/Main_Page, 2009.
[8] E. Szoka, "Earth Sculptor," http://www.earthsculptor.com/, 2009.
[9] Autodesk, "Maya," http://usa.autodesk.com/adsk/, 2009.

Hollie Boudreaux received the B.S. degree (summa cum laude) in computer science from Nicholls State University, Thibodaux, LA in 2004, and the M.S. degree in computer science in 2007 from the University of Louisiana at Lafayette, Lafayette, LA, where she is currently working toward the Ph.D. degree. Her research interests include computer graphics, video game design and development, artificial intelligence, and multiagent systems.

Jim Etheredge received the M.S. degree in computer science from the University of Southwestern Louisiana in 1986 and the Ph.D. in computer science from the University of Southwestern Louisiana in 1989. He is currently an associate professor of computer science at the University of Louisiana at Lafayette, Lafayette, LA and the coordinator for the Video Game Design and Development concentration of the undergraduate computer science curriculum. His research and teaching interests include video game design and development, artificial intelligence, multiagent game systems, and database management systems.

Ashok Kumar is an Assistant Professor in the Department of Computer Science at the University of Louisiana at Lafayette. Dr. Kumar obtained his Ph.D. in 1999 and worked for four years in industry before joining academia full time. He has over fifty publications in refereed journals, conferences, and book chapters. He has served on the program committees of several conferences.