A Pedagogy-Oriented Modeling-and-Simulation Environment for AI Scenarios

Dan Tappan
Department of Computer Science
Idaho State University
921 S. 8th Ave., Stop 8060
Pocatello, ID, 83209-8060

Abstract - **Programming assignments are an effective way for students to investigate many important and fun aspects of AI. However, for any task of reasonable breadth and depth, especially involving complex, compositional agents, actions, and graphics, most of the programming effort goes into mundane, tedious, time-consuming, and error-prone administrative aspects. Moreover, time constraints usually result in designs that are not extensible or reusable for subsequent assignments, which repeat this overhead throughout the semester. This pedagogy-oriented modeling-and-simulation framework provides convenient support capabilities to get students quickly playing with a wealth of agent-based AI content. It contains extensive, highly configurable, yet user-friendly, engineering, physics, and communication models for arbitrary components and task environments. These components are managed automatically in a stochastic Monte Carlo simulation that allows students to define, test, and evaluate their quantitative performance for a wide range of controlled experiments.**

INTRODUCTION

Student projects in AI can be fun, exciting, and educational. However, most programming tasks involve a disproportionate amount of time-consuming, error-prone administrative code that distracts students from the true focus. This pedagogy-oriented modeling-and-simulation system provides a comprehensive framework of highly extensible support functionality to investigate many concepts and strategies in AI and intelligent control systems. In particular, it facilitates building, manipulating, and observing arbitrary intelligent agents in a broad range of operational contexts. It also helps foster an understanding of proper methodology in designing, executing, and evaluating controlled experiments [1,2]. Most class projects are heavy skewed toward producing a solution, at the expense of understanding how it works, how well it works, how it might be improved, and so on [3]. This system tries to balance this synthesis aspect with corresponding analysis that formally and quantitatively demonstrates performance. Learning is an iterative process, but without the analysis aspect, an integral part of the feedback mechanism is missing [4].

BACKGROUND

Toolkits, application programming interfaces, software development kits, game engines, and other variants are not new to AI. This system does not claim to be a revolutionary advancement on existing systems, but it does strive to unify many of their essential aspects into a pedagogical framework that forces students to focus primarily on the AI content, and not so much on the support programming. It is also heavily oriented toward careful rigor and discipline in design, implementation, testing, and evaluation. Ultimately, it is the students who provide the intelligence to their solutions. They must therefore truly understand what their solutions can and cannot do, and what they are asking them to do. Hacking—the basis of most programming assignments—does not result in good code, good agents, or good learning [5].

The pedigree of this system attests to the success of its philosophy. It derives—as a complete redesign and reimplementation—from two versions of a large-scale, agent-based modeling-and-simulation system developed for the U.S. Army in support of its Future Combat Systems program [6,7]. In the first version, the inexperienced team produced a monumental (yet admittedly successful) hack. The second version, built within this rigorous framework and philosophy, resulted in multiple awards and the first accreditation of such an analysis tool by the Army in over 20 years. The code was good, the agents were good, and the team learned about how and why they succeeded.

There are many related systems with varied backgrounds and goals. Gaming applications, for instance, have embraced the powerful role of AI in realistically driving the behaviors of computer-controlled characters. Autodesk Kynapse, AI Framework, and the FEAR software development kit, for example, integrate with game and graphics engines for a rich, multimedia experience. In a classroom-oriented pedagogical role, however, their learning curve can be overwhelming. Other work like AIspace, AI Toolkit, Prometheus, OpenSteer, Game::AI++, Soar, the InExIn library, and Agent Development Kit provide implementations for a wealth of AI approaches at a lower, somewhat standalone level. This work plays an intermediate role like breve, MASON, NetLogo, and Swarm. In particular, its emphasis is to provide a packaged, student-oriented environment in which to design, test, and evaluate agent-based systems, as well as to facilitate disciplined software design in AI systems. Sloman [8] discusses many related considerations that were incorporated into this work.

This system is entirely Java-based. Java 3D provides the visualization, and JavaCC parses the support files, most of which are defined as XML. This organization supports

consistency and portability in deployment and interaction with other tools. The design is also meant to be reasonably lightweight to support the older, less powerful computers that many students have. This consideration is especially important for use in public schools, which tend to be significantly behind the hardware upgrade curve.

MODELING

Within the context of this system, modeling refers to defining the composition and behavior of the agents and their environment. It encourages disciplined forethought in this organization, which the system later strictly enforces to ensure that everything plays by the rules.

A. Environment

The virtual world in which agents operate is currently a simple, passive element with no interaction between it and the agents. It is designed primarily for open-field, outdoor scenarios of arbitrary scale in either two or three dimensions. It supports only flat-earth topography, although variable terrain will be available in later versions.

B. Agents

Agents are the core of the model. The system uniformly accommodates both natural and artificial variants, like animals and machines, respectively. They may play active or passive roles to interact individually or collectively with respect to cooperation, competition, and so on. The underlying behavior and its implementation are entirely the choice of the student designer. In support of these choices, the system provides a rich set of elements to reduce the workload. In particular, it takes advantage of well-established design patterns that divide agents into their structural, behavioral, and creational elements [9]. It also provides a wealth of miscellaneous supporting functionality.

Structural elements define the composition of agents. They address physical realities of assembly, as well as virtual aspects that help the designer observe and interpret performance.

Physical components are literally the recursive building blocks of an agent. Each is a rectilinear box defined by a unique identifier and width, depth, and height dimensions. Fig. 1 (a) shows a simple tank decomposed into a hull, turret, barrel, and sensor. A component also has various appearance attributes like color, transparency, texture, and wireframe representations. Transparency is especially useful for representing both actual and believed position and attitude in belief-desire-intention (BDI) models [10,11].

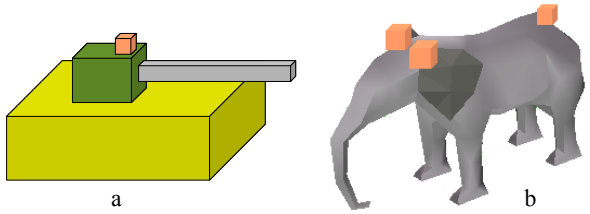

Fig. 1: Compositional Agents

The box may also be rendered as an externally defined three-dimensional model with arbitrarily complex polygons, such as the elephant with three attached "sensors" in Fig. 1 (b). Many models in the supported Wavefront™ and 3D Studio™ formats are freely available on the web. Google also provides a substantial library of compatible models for its SketchUp™ software, which is available for free evaluation. Regardless of the model, the underlying geometry is still crudely bounded as a box.

The compositional structure of an agent is based on engineering gimbals. This mechanism requires that each component have a single female socket positioned somewhere relative to its origin. The socket may have constraints that define and limit its degrees of freedom (DOF), as well as advanced properties like latency, speed, and acceleration. The socket allows another component (the supercomponent) to mount this component (as its subcomponent) to itself through a male ball, which is relative to the origin of the supercomponent. A component can have an unlimited number of balls, allowing for unbounded compositionality and degrees of freedom.

Managing these essential physicalities invariably causes students tremendous grief. Any practical implementation (such as the quaternions used here) is decidedly nontrivial and far beyond the scope of most class projects. To mitigate this problem, a separate DOF manager for each component provides numerous features, for which students merely need to define the constraints. This declarative approach frees them to focus on *what* the agent is supposed to do, and not so much on *how* their code needs to make it happen. This manager supports the two common DOF systems in Fig. 2, which reflect how most natural and artificial real-world components articulate.

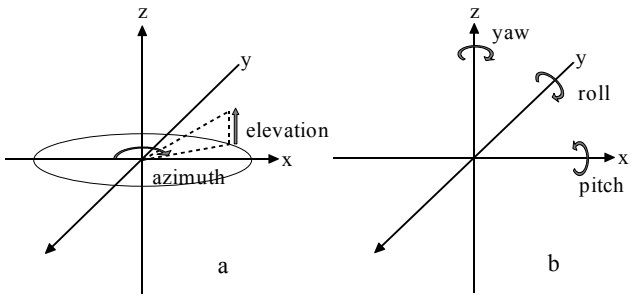

Fig. 2: Degree-of-Freedom Systems

The 5-DOF system (a) uses azimuth and elevation for components like a gun barrel, whereas the 6-DOF (b) uses yaw, pitch, and roll for components like an airplane. In both cases, the three degrees of freedom for position (x, y, and z) are identical. The choice of system is important because student code needs to know how to manipulate its commutativity properly. In the 5-DOF system, the two degrees of attitude freedom are independent of each other, and either update order results in the same configuration. In the 6-DOF system, however, there are six possible orders, each of which may result in a different configuration. Students are not

likely to be aware of such design considerations for a physical agent, but this approach makes them apparent.

The gimbals recursively account for any physical interconnections and dependencies between components. Each agent has a required base component, such as the hull of the tank in Fig. 1 (a), as well as optional subcomponents, like the turret on the hull, which in turn has sensors and actuators (i.e., the gun barrel). Changing the configuration of any degree of freedom on any component automatically propagates the corresponding changes throughout all subcomponents and satisfies any dependencies. As a reality check, this process also verifies that student code does not force the mechanical system into an invalid configuration, which is common when students do not truly understand what they are modeling and why.

Sensors play a key role in acquiring percepts from the environment in most agent-based systems [11]. Sensor components can be fixed, where they only move dependently with respect to their supercomponents, or they can also move independently. In either case, they support a horizontal, vertical, or combined field of view (FOV), which, in the simplest form, helps determine whether another agent (or component) is within an angular wedge (for two dimensions) or a pyramidal frustum (for three dimensions) [12]. Movable sensors can adjust the FOV within the angular limits of a field of regard (FOR) to support scanning; e.g., moving a head back and forth. The timing service, to be discussed shortly, can automatically manage this movement, or student code can perform it manually.

Optional, advanced sensor functionality accounts for noise, performance degradation over range, or as Fig. 3 illustrates, probabilistic preferences for central versus peripheral vision in an FOV. The probability of false positives and negatives is also configurable. Ultimately, student code needs to decide how to process these percepts, of course, but the system makes it relatively easy to acquire them in ideal or degraded forms.

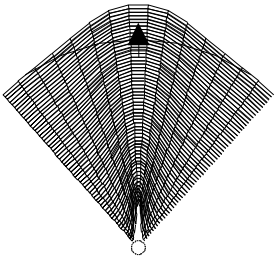

Fig. 3: Field-of-View Preference Distribution

Unlike physical components, which play an active role as the backbone of an agent, virtual components do effectively nothing. Their role is to provide passive metainformation that supports a simulation, but that the model itself cannot use. These components can be connected to any physical component through fixed gimbals. They do not allow recursive decomposition.

There are currently three types of virtual components, as illustrated with bees in Fig. 4: two-dimensional triangles (a) or three-dimensional frustums are typically used to render a translucent FOV so students can see where a sensor is looking; polylines (b) serve as reference markers for imaginary boundaries, and so on; and labels (c) identify agents and their components.

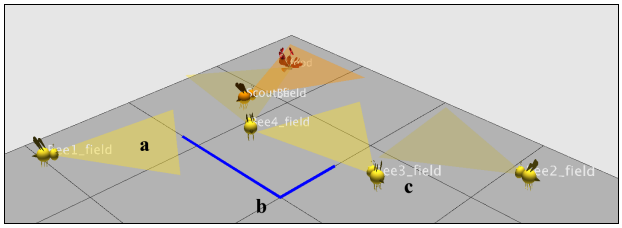

Fig. 4: Virtual Components

Behavioral elements overlay the compositional structure, such that students must first define *what* an agent is, then *how* the agent functions within these constraints. This philosophy enforces a prescribed chain of responsibility for delegating the execution of requests to the appropriate components within an agent and between agents. It also discourages undisciplined programmatic "cheating," where the model uses information or performs actions that it rightfully should not be able to access. For example, a sensor usually relies on percepts to determine where another agent is based on defined limitations like acuity. Unfortunately, its code can easily access another programmatic object (intentionally or not) through direct method calls to get the actual data it needs, as opposed to indirectly deriving the believed data. Such a tempting shortcut undermines the purpose of modeling the sensor, though, and it probably skews the results of the simulation, as well.

Behavioral elements cannot prevent students from implementing such dubious solutions, but at least they can make such approaches more apparent for grading and discussion. The overarching design philosophy of this system is to build agents that faithfully model the real world and map back to it. The expectation is that there is bidirectional correspondence between code and its real-world counterparts; therefore, discrepancies need to be examined and justified carefully. Students need to learn firsthand how to recognize these cases, and then how to justify them persuasively, because most design decisions in computer science are open-ended, subjective, and often controversial [13,11].

Behavioral elements are a conglomeration of at least the *composite*, *strategy*, *chain of responsibility*, *mediator*, *state*, *observer*, *command*, and *interpreter* design patterns [9]. The code itself also serves as a strong metaexample of a disciplined design and architecture. The primary behavioral element is a communication network with protocols for intra- and inter-agent message transfer. These protocols greatly reduce the complexity of managing intercommunication, while also reducing the temptation for programmatic cheating and outright hacking at a solution.

A message is typically an event, a request for data or action, or a response. It consists of the sender and recipient identifiers and an optional, arbitrary payload. Recipients can be individual components, classes of components (e.g., all agents on a team), or unrestricted broadcasts to all

components. A message allows itself to be handled in various ways (abridged here) depending on the recipient and its state:

- `IGNORED`: a recipient determines that a message (usually a broadcast) is not intended for it or is irrelevant, so it silently discards it.
- `ACCEPTED_PROCEED`: a recipient determines that it can immediately act on a message and return any result. The sender can proceed after receiving it.
- `ACCEPTED_WAIT`: a recipient determines that it can act on a message, but the result will be forthcoming after an indeterminate delay. A confirmation of this condition is returned in the interim so the sender knows to expect the result eventually, and it can proceed with other activities (if possible) until then. It can also cancel the message and reissue it elsewhere.
- `REJECTED_DISCARD_INFORM`: the recipient would otherwise be able to act on a message, but it cannot at the moment. This condition is returned to the sender, which can then decide how to handle it. A `REJECTED_DISCARD_SILENT` variant omits the return, which is similar to instructing the post office to abandon a package that cannot be delivered to avoid the return postage.
- `REJECTED_RESUBMIT_INFORM`: the recipient would otherwise be able to act on a message as described above. It will inform the sender when it is free to accept the message again, at which time the sender must decide whether to reissue it. In the meantime, it could also reissue the message elsewhere.
- `REJECTED_RESUBMIT_AUTO`: the recipient would otherwise be able to act on a message as described above. It holds onto the message until it can process it, at which time it informs the sender of either `ACCEPTED` state above.

Timing and synchronization of events, especially in stochastic, multiagent simulations, can be overwhelming programming tasks for students. Another behavioral element allows components and their managers to subscribe to arbitrary updates. It conveniently supports fine-grained actions that would otherwise be difficult to coordinate. For example, slewing the azimuth of a sensor from 0 to 90 degrees over 10 seconds would require substantial student code to manage reasonably smooth movement. Typical approaches are inconsistent and often lead to bizarre "relativistic" effects where multiple agents operate under different time systems. As time is a shared phenomenon, it is best handled globally by this service, not locally by agents. In this case, the operational approach to slewing must also be carefully considered. If the code simply increments the angle by 90 degrees after a 10-second wait, then the sensor never passes through any intermediate angles, and it may not perform as expected. This service allows subscribers to specify in intuitive ways how they want to be updated; for instance:

- *n* times over *s* seconds
- every *s* seconds over *t* seconds
- every *s* seconds *n* times
- every *s* seconds continuously until canceled
- one time for *s* seconds, then self-cancels

The sensor could subscribe to be uniformly updated six times over 10 seconds, and at each update, it would increment its azimuth by 15 degrees. Timing behavior is thus appropriately delegated to the system, while sensor behavior rightfully belongs to the sensor itself. Such disciplined responsibility, cohesion, and coupling are key to effective, understandable software [13]. Typical, *ad hoc* attempts at localized time management can be a nightmare to debug and maintain. Not only do they undermine the goal of a simulation, but they frustrate students and may turn them off to the subject matter.

Other built-in services manage specialized processes like collision detection and localized gravity (for outer-space scenarios). They are expensive and usually unnecessary, so only components that need to participate should subscribe. Finally, the plug-and-play structure of the system allows students to implement other services as needed.

Creational elements manage the creation, assembly, disassembly, and destruction of agents and their components. They use *prototype*, *flyweight*, *factory*, and *builder* patterns to simplify this process so students simply request a new agent as necessary [9]. The underlying XML definition of its composition is analogous to the data definition of a class in object-oriented programming, and an agent is likewise an instance of it. Agents can be permanent, such as walls, semi-permanent, such as players that can be killed, or temporary, such as projectiles that fly out and expire after a collision or a certain range or time (which is managed by a behavioral service). Instantiation is mostly a rubber-stamp process with minor variation, such as assigning unique identifiers.

Support elements provide a variety of miscellaneous functionality. Their code is not necessarily difficult to implement, but it can be tedious, time-consuming, and error-prone [14,12]. From overwhelming anecdotal evidence, it is clear that students spend an inordinate amount of time on such peripheral aspects of their projects, at the expense of the intended content.

These elements derive from a common data type that mitigates many of the typical problems students encounter with units, magnitudes, conversions, and so on. The algebraic validity of many results can also be verified. Most elements fall into the following categories:

- *Situational awareness*: absolute and relative coordinates as (x, y, z) and (longitude, latitude, altitude); absolute and relative attitudes as (azimuth, elevation) and (yaw, pitch, roll); coordinate collections for lines, planes, and boxes.
- *Differential awareness*: static bearings (angle and range) and dynamic bearings (with velocity) from one agent to another.
- *Motion*: speed, velocity (horizontal, vertical, combined), time, acceleration, drag.

- *Vision*: rays for line-of-sight determination; atmospheric attenuation.
- *Path-following*: smooth, connected lines with spline interpolations and attitude correspondence.
- *Engineering and physics*: frequency, wavelength, duty cycle, intensity, reflectivity, cross-section, response curve, noise, signal-to-noise ratio, false-alarm rate, attenuation, temperature.
- *Coordinates*: conversions between mathematical, navigational, and graphical coordinate systems, which cause students endless trouble.
- *Miscellaneous*: up/down counters with trigger subscribers; probability.

Simulation

Within the context of this system, simulation refers to putting the model into operational scenarios within the environment to observe and measure its performance with respect to well-defined tasks. The simulation facilitates rapid, iterative refinement of approaches, where students run their code, observe how it performs, and improve it. The simulation framework is based on a Monte Carlo methodology for controlled experiments that can convincingly demonstrate the value of improvements and overall performance [15].

A. Control Simulation

The control simulation establishes a baseline performance measure against which to compare the results of subsequent changes in the test simulation. This step mitigates one of the most common problems with projects: most student effort is spent on the synthesis of a solution, and any analysis of it is minimal. In fact, anecdotal evidence shows that most students subjectively decide to stop when their solution appears to work at all, and there is rarely an objective measure of its performance. Consequently, students may not develop the skills to recognize what needs improvement, and then how to test how well their improvements actually work.

B. Test Simulation

The test simulation measures differences in performance with respect to the control simulation. For instance, the task of an agent might be to locate something with its sensor. The control simulation, using one type of sensor, has an inherent quantitative performance, but no benchmark against which to assess its meaning or significance. The test simulation, using a different type of sensor, and differing *only in this respect*, likely produces different results. The difference between the two can be attributed directly to this one and only change. If the change was an improvement in performance, then it suggests a promising trajectory toward further improvements. Likewise, a negative or inconsequential change suggests trying another approach. Such iterative refinement allows students to see the cause-and-effect relationships of their decisions.

Components are carefully controlled within the simulation to ensure that modifications to them have no side effects, or at very least, the side effects are known. This stochastic stability is critical because having more than one simultaneous change may confound the analysis and undermine any conclusions. It is unreasonable to expect that students could manage this task themselves due to its complexity.

Visualization and Analysis

Agent behavior is usually entertaining to watch. Observation is also a useful, qualitative measure of performance, as students can often immediately see whether their choices lead to the expected results. To this end, a three-dimensional visualizer provides graphical rendering of the execution of a simulation. It functions in two modes. As a standalone application, it does not need to be tied to the rest of the system. In fact, the source of the data to render is irrelevant, provided that it is in a text file in the correct format. This feature allows other applications in other languages to generate data. As an integrated application, interaction is possible. Students can modify (to a limited degree) the configuration of the world while the simulation is running; e.g., push a cube in front of an obstacle-avoiding robot. A variety of metainformation can be depicted, such as identifiers, positions, attitudes, vectors, distances between agents, paths followed, and propagation of rays. Fig. 5, for instance, shows the "breadcrumb" tracks of an unmanned aerial vehicle as it moves through the world. Superficial inspection of such output is often sufficient for grading.

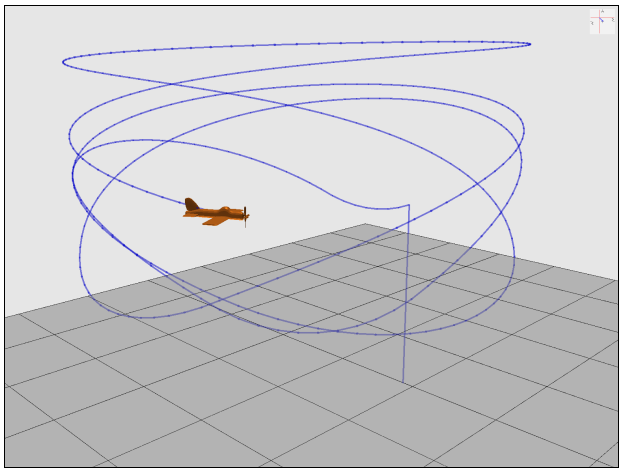

Fig. 5: Visualization

In either mode, the simulation can be played like a movie, but dynamically from any vantage point. Such visualization capabilities allow instructors and students to demonstrate solutions in the classroom, where everyone can contribute to the evaluation and comparative grading. In addition, written reports and small presentations with representative screen shots that address how problems were solved can help improve students' communication skills. Finally, the visualization serves as an appealing demonstration tool for recruitment events in computer science, which often lack a "coolness" factor to attract interest [16].

In addition to visualization, nearly all internal data can be exported to applications like Excel™, MATLAB™, and

gnuplot for external quantitative analysis. Determining which data to capture, and their significance, is the student's responsibility as part of the learning process.

Evaluation

Formally evaluating a modeling-and-simulation, test-and-evaluation system like this one is a somewhat circular process. A reasonable measure of success is how well it has integrated into a number of undergraduate courses. In particular, it has been the core software for two offerings of an upper-division AI course, with a total of 16 programming assignments. It also supported static and dynamic examples in many lectures. Some nature-inspired examples are herding, flocking, swarming, chasing, collision avoidance, communication, vision, echolocation, and predator-prey scenarios; some engineering-oriented ones are autonomous vehicle navigation, robotic-arm manipulation, and game-based detection, tracking, prediction, and engagement.

Under a different guise (downplaying the AI aspects), it has also contributed to several offerings of two lower-division courses in advanced object-oriented design, as well as played a supporting role in an upper-division software-engineering course by demonstrating how the system itself is designed.

Ultimately, however, it is the students who decide whether any software helps or hinders their educational experience. In this case, anecdotal evidence and formal course evaluations suggest that it has been received well. A significant amount of work is still needed to polish it for widespread distribution and support. Nevertheless, it has shown itself to be a worthwhile contribution to design-related pedagogy, whether AI-oriented or not.

Future Work

There are two facets to future work with this system. Within the existing framework, a graphical user interface, possibly in the form of an integrated development environment, is necessary to unify the various, disjoint components of the system. For example, the complex XML files for data and configuration are currently onerous to create and modify. This limitation conflicts with the goal of a friendly rapid-prototyping system. Another extension is to add triangles to the current rectilinear definition of components. Triangles will support higher-fidelity abstraction of three-dimensional models, as well as variable terrain. Many AI problems, especially for practical, real-world ones like variants of the DARPA Grand Challenge, need a richer, non-flat environment [17].

Beyond the existing framework are potential spin-off applications. The current system is targeted to the university level. Younger audiences could also benefit from it, however, because they tend to show great interest in robotics [16]. It would be excessively complicated and overwhelming for them to use, however, so one spin-off is a simplified facade around the system, such that the basics of defining and running simple agents are easily accessible. The focus would be fun and logical thought, not formal evaluation. For advanced children, especially those participating in competitions like the FIRST Lego League robotics challenges, some analysis could be practical, though. For example, there are many options to consider when designing a robot. Teams tend to fixate on one idea only. With some analysis appropriate to their level of understanding, multiple designs could be investigated and evaluated before committing to any construction.

Conclusion

This system provides an extensive, freely available software environment with a large set of commonly needed support features for designing, implementing, testing, and evaluating models for AI and related engineering problems. Its goal is to mitigate much of the mundane, tedious, time-consuming, and error-prone administrative overhead involved in programming AI solutions, so students can focus more on the AI and less on the support code. It executes student-defined models in a Monte Carlo simulation that generates and records stochastic data for offline analysis. In various incarnations, it has been received well over several semesters for courses in AI, object-oriented design, and software engineering.

References

[1] Z. Dilli, N. Goldsman, J. Schmidt, L. Harper, and S. Marcus. "A New Pedagogy in Electrical and Computer Engineering: An experiential and conceptual approach": in Proceedings of Frontiers in Education, pp. T2C3-7, Boston, MA, 2002.

[2] T. Wiesner and W. Lan. "Comparison of Student Learning in Physical and Simulated Unit Operations Experiments", Journal of Engineering Education, Vol. 93, No. 3, pp. 223–231, 2004.

[3] D. Tappan. "ShelbySim: A Transparent, Pedagogy-Oriented Simulator for Computer-Based Systems", International Journal of Engineering Education, in press.

[4] R. Irish. "Engineering Thinking: Using Benjamin Bloom and William Perry to Design Assignments, Language and Learning Across the Disciplines", Vol. 3, No. 2, pp. 83-102, 1999.

[5] L. Tong. "Identifying essential learning skills in students' engineering education": in Proceedings of HERDSA 2003, Canterbury, New Zealand, 2003.

[6] J. Engle. "AMSAA's SURVIVE Model plays key role". RDECOM Magazine, August 2005.

[7] D. Tappan and J. Engle. "The AMSAA SURVIVE Model": in Proceedings of U.S. Army 16th Annual Ground Vehicle Survivability Symposium, Monterey, CA, 2005.

[8] A. Sloman. "What's an AI Toolkit For?" AAAI Technical Report WS-98-10, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[10] M. Jones. "Artificial Intelligence: A Systems Approach". Infinity Science, 2008.

[11] S. Russell and P. Norvig. "Artificial Intelligence: A Modern Approach". Pearson, 2003.

[12] D. Bourg and G. Seemann. “AI for Game Developers”. O’Reilly, 2004.

[13] R. Pressman. “Software Engineering: A Practitioner’s Approach”. McGraw-Hill, 2010.

[14] D. Bourg. “Physics for Game Developers”. O’Reilly, 2002.

[15] G. Fishman. “Monte Carlo: Concepts, Algorithms, and Applications”. Springer, 1995.

[16] P. Jonsson. “Can competitions raise ‘cool’ factor of math, science?” Christian Science Monitor, 17 May 2008.

[17] M. Montemerlo, S. Thrun, H. Dahlkamp, D. Stavens, and S. Strohband. “Winning the DARPA Grand Challenge with an AI Robot”: in Proceedings of American Association of Artificial Intelligence, Boston, MA, 2006.