A Pedagogical Framework for Modeling and Simulating Intelligent Agents and Control Systems

Dan Tappan

College of Engineering, Idaho State University
921 S. 8th Ave., Stop 8060
Pocatello, ID 83209-8060
tappdan@isu.edu

Abstract

Classroom assignments are an effective way for students to investigate many important (and fun) aspects of artificial intelligence. However, for any project of reasonable breadth and depth, especially involving multiple agents and graphics, most of the programming goes into tedious, error-prone administrative tasks. Moreover, time constraints usually preclude an extensible and reusable design, so this overhead repeats itself throughout the semester. This pedagogy-oriented modeling-and-simulation framework provides all the necessary support capabilities to get students up to speed quickly on playing with AI content. It contains extensive, highly configurable, yet user-friendly, engineering, physics, and communication models for arbitrary components within a definable task environment. These components are managed automatically in a stochastic simulation that allows students to define, test, and evaluate their performance over a wide range of controlled experiments.

Introduction and Background

Student assignments and experiments in artificial intelligence can be fun, exciting, and educational. Most programming assignments, however, involve a disproportionate amount of mundane, administrative code that distracts students from the AI focus. This pedagogy-oriented system provides a straightforward framework of highly extensible components and functionality to investigate many concepts and strategies in AI and intelligent control systems (Russell and Norvig 2003, Bourg and Seemann 2004). It also helps foster an understanding of proper methodology in the design, implementation, testing, and evaluation of formal experiments.

This system is a lightweight version derived from a full-scale, accredited simulator developed for the Department of Defense to evaluate components for the U.S. Army's Future Combat Systems program (withheld, Engle 2005). Another derivative has also been used successfully in AI for interpreting spatial relations in natural language processing (withheld). For portability, the system is written entirely in Java, with Java 3D as the display engine.

Modeling

Within the framework of this system, *modeling* refers to defining what the entities to study are and what they can do. The *simulation* counterpart then puts them into operational contexts for formal evaluation.

Environment

The environment is a three-dimensional world of arbitrary scale. As fancy graphics are not the focus, the working area is normally depicted as the tabletop in Figure 1. The viewer may interactively move to any vantage point throughout a simulation, as well as click on entities to query them about their underlying details. The terrain model currently supports only this flat surface, but vertical relief may eventually be incorporated. The default physics model provides basic, global support for naive kinematics like velocity, acceleration, and gravity (Bourg 2002). Students can configure it in various ways or substitute their own plug-and-play models.

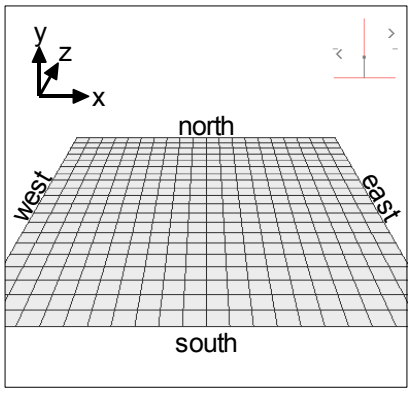

Figure 1: The World

Components

Components are hierarchical building blocks. *Static* components passively occupy space. They typically serve as immobile obstacles to navigation or visibility. *Dynamic* components interact with the world by generating and/or responding to events. A component is physically a three-dimensional box, with *height*, *width*, and *depth* properties, that may recursively connect to any number of subcomponents through one of two interconnection models:

- The *6-DOF* model in Figure 2 provides six degrees of freedom (DOF): *x*, *y*, *z* coordinates for position, and *yaw*, *pitch*, and *roll* angles for the three attitude axes. It is the preferred model for most components because it provides the most flexibility to move and face anywhere.

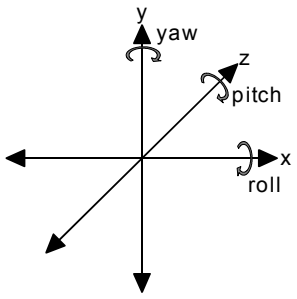

Figure 2: 6-DOF Model

- The *5-DOF* model in Figure 3 provides the same position representation, but it uses *azimuth* and *elevation* angles for only two attitude axes. It is useful for aimable components like sights and gun barrels.

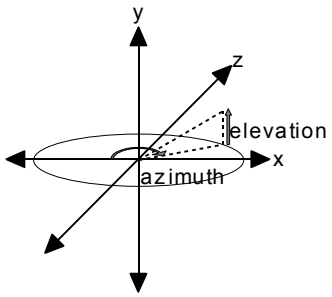

Figure 3: 5-DOF Model

The format for specifying and reporting any degree of freedom supports two interconvertable systems:

- *Absolute* angles reference the context-free, fixed coordinate system of the world. For example, north is always 0° yaw or azimuth.
- *Relative* angles reference the context-sensitive, variable coordinate system of each component (and/or its supercomponents). For example, 0° yaw or azimuth is in front of a component, whereas +45° is to its front-right. Similarly, if the supercomponent is pitched +10° (up), and the component is pitched -10° (down), the resulting pitch of the component is 0°.

The interconnections, once initially defined by the students, are managed automatically. For example, updating the position and/or attitude of the base entity, say the hull of a tank, correspondingly updates its turret, which subsequently updates the gun barrel. The underlying mathematical model uses quaternions, which are an interesting modeling topic in their own right. This code is fully commented and available for student inspection.

Either model supports optional constraints to limit the minimum and maximum range of each degree of freedom. For example, the gun may have ±10° travel in pitch, and 0° in azimuth. Thus, in order to change the azimuth of the gun, the turret, on which it is mounted, must be updated. The physics model manages the optional velocity and acceleration constraints.

One of the most useful applications of composite, articulated components is for sensors, which receive information about the environment. The basic form consists of two parts:

- A *field of view* (FOV) in Figure 4 is a pyramidal frustum, defined by height and width angular limits, that projects relative to a definable origin on the component. Anything within the frustum is considered to be in view of the sensor. Additional constraints like obstacles to line of sight and degradation of acuity over distance are handled separately.
- A *field of regard* (FOR) specifies the limits over which the FOV can move, if it is configured as a moving sensor. Together, the FOV and FOR support scanning. A real-world example is using binoculars to locate a target: the FOV is narrow, and it must be updated across the FOR, typically through restricted neck movement.

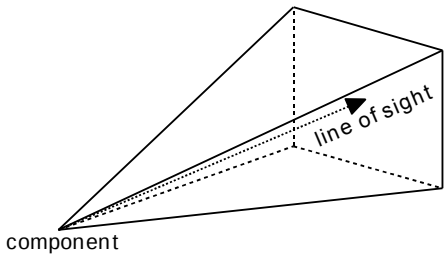

Figure 4: Frustum

This built-in physics and engineering functionality frees students to focus on the AI aspects of their projects. For example, one course assignment used two or more sensors as eyes, in combination with rudimentary trigonometry, to emulate stereoscopic vision for range-finding. The students discovered the law of diminishing returns for more than two sensors. Similarly, they investigated different arrangements of two sensors to model the pros and cons of how various animals view the world.

Updating can be configured to occur on demand or automatically (e.g., initially clockwise at 10° per second until the limit is reached, then the reverse, repeated a configurable number of times, including indefinitely). This feature eliminates the need for students to manage the extensive complexities of initiating, terminating, synchronizing, and timing actions in their own code.

Sensor configurations can also automatically account for simple, real-world, physical limitations. For example, acuity may degrade as a configurable function of distance, thereby making far-away objects in field of view harder to see than closer ones. This information couples with probability-based support functions to produce false positives (seeing something that is not there) and negatives (not seeing something that is there). Such rich capabilities support powerful analyses of system performance and reliability.

Agents

The primary role of components is to acquire raw information about the environment. They may also execute student-supplied code to preprocess it; e.g., sorting objects by distance and filtering some out. However, they are generally "dumb" engineering devices in that they do not make high-level decisions based on their localized, contextually weak information. Agents play this global role as a hierarchical collection of intercommunicating components. It is here that students normally investigate the AI aspects of their designs.

This command-and-control framework is based heavily on established design patterns in object-oriented programming (Gamma et al. 1995). Effective engineering control systems (intelligent or otherwise) usually reflect a purposeful, well-structured design. In software engineering, however, students (and professionals) tend to build haphazard contraptions that bear little resemblance to the real-world counterparts they are modeling. This system is designed to foster design-related self-discipline. In particular, its agent framework natively supports the following design patterns for students to augment:

Structural patterns define what entities are:

- The *Composite* pattern, as described earlier, supports hierarchical structures of components.
- The *Flyweight* pattern supports multiagent simulations of many entities and/or of entities with complex, shared hierarchical structures. It reduces memory use and simulation overhead, thereby improving simulation speed.
- The *Facade* and *Decorator* patterns allow baseline entities to play similar, reusable roles with minimal additional code or modification. For example, a car and truck are basically subtypes of a common vehicle that is qualified as passenger-carrying and cargo-carrying, respectively. In all other respects, they may be the same.

Creational patterns define how to build and configure the structure of entities:

- The *Prototype*, *Factory*, and *Builder* patterns provide an object-oriented production mechanism within the simulation. For example, a student might specify for a traffic simulation that there are any number of cars and trucks with certain, unique properties. Setting up a simulation should not require substantial coding because these patterns do the lower-level assembly work.

Behavioral patterns define what entities can do:

- The *Strategy* pattern supports the reusable, plug-and-play philosophy of components. It allows students to swap different implementations under investigation to determine their comparative performance.
- The *Chain of Responsibility* pattern supports disciplined engineering considerations in the interconnection of components. At the heart is delegating tasks to the proper components, then enforcing an established chain of command for information transfer up and down the hierarchy. It reduces the opportunity for spaghetti code and hacks, and it makes such inferior solutions apparent.
- The *Visitor* pattern serves as the transmission medium within the chain of command. It can be modeled with real-world engineering considerations, such as bandwidth, speed, latency, noise, and so on.
- The *Command* and *Interpreter* patterns serve as the intercommunication content model to allow events and decisions to propagate appropriately among entities. For example, one course assignment had a centralized traffic controller instructing agents to avoid collisions through different classes of English-based commands: absolute ("turn to 0 degrees"), quantitative relative ("turn left by 90 degrees"), and qualitative relative ("keep turning left until I say stop").
- The *Observer* and *Mediator* patterns serve as the intercommunication delivery model to pass commands between entities. They support variations on one-to-one and one-to-many transmissions of arbitrary, student-defined content.
- The *State* pattern provides a straightforward decision-making framework for implementing finite-state processes with minimal code. Students can focus on the content and meaning of the processes, instead of on their implementation.

Simulation

A stochastic, discrete-event simulator manages all entities and serves as a test-and-evaluation framework. Students first design, assemble, and configure their components and agents, then they place them into an operating context of

the environment. The intent is to run controlled experiments to measure performance according to students' criteria. This process consists of two parts:

- A *control* simulation establishes baseline performance; e.g., tracking and intercepting a target in a predator-prey simulation.
- A *test* simulation augments the baseline by changing one—and only one—aspect of it; e.g., increasing the magnification of the predator's eye sensors, or replacing them with a different model. Any measured performance differences between the baseline and test simulations can therefore be directly attributed to this perturbation of the model. In other words, it establishes a cause-and-effect relationship.

In both parts, a general-purpose logger records standard details (like time, positions, attitudes, and events), as well as those specified by the student. The logs export as plain text files.

A major strength of stochastic simulation lies in its probability-based non-determinism. A single run is therefore meaningless from an analytical standpoint: the results could be representative of reality, or they could be purely coincidental, and there is no way to assess any confidence in either outcome. The simulation framework allows students to run an arbitrary number of independent iterations (often thousands) with the same initial conditions so that the probabilities unfold naturally over their inherent, yet hidden, independent and dependent distributions. The logger keeps track of the individual and aggregate results.

Analysis

Analysis is an external process involving the premise of the experiments, their results, and the students' qualitative and quantitative reasoning abilities to process the data and draw conclusions. Consistent with the overall philosophy of this work, the students are freed from most of the mundane, tedious overhead that distracts them from the focus. The analysis work, however, is entirely their own, and is done with external tools like spreadsheets and statistics packages. This dovetails well with the scientific method of running experiments to prove (or disprove) and explain hypotheses. While it certainly possible that students can randomly generate and test (in other words, hack) until something acceptable emerges, with sufficient pedagogical emphasis from the instructor on experimental discipline and rigor, students should learn firsthand the value of well-defined experiments.

Discussion and Future Work

In Spring 2007, an earlier proof-of-concept form of this system was successfully fielded in an upper-division undergraduate AI course. The encouraging results, as well as student feedback, contributed to its continual development and improvement. It is expected to play the core computational role in the next offering of this course. Once stable, it will be made available in the public domain for academic use.

References

<withheld-a>

<withheld-b>

Bourg, D. 2002. *Physics for Game Developers*. O'Reilly, Sebastopol: CA.

Bourg, D. and Seemann, G. 2004. *AI for Game Developers*. O'Reilly, Sebastopol: CA.

Russell, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Pearson, Upper Saddle River: NJ.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis: IN.

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