Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games

Daniel Fu and Ryan Houlette, *Stottler Henke Associates*

A few years ago, we attended a conference bringing artificial intelligence researchers and game developers together. Having just begun enhancing our company's suite of AI tools for simulation development, we were ready to be enlightened. AI researchers were going to offer up the latest technological advances, while the game developers were going to wow us with their tried-and-true techniques for authoring game AI. While the conference was enlightening, what we found most surprising was that game AI developers almost always implemented their AI from scratch. Every game was just too unique and so resisted reuse of existing representation and code. In fact, several developers openly stated that a general "AI toolkit" couldn't even exist. What's even more surprising is that these opinions were expressed even though much of the work in game development focuses on creating authoring tools. Given that most games adopt standard AI techniques that were solved decades ago, why hasn't there been a concerted effort to create a standard way of articulating AI?

Pessimism aside, research, game, and simulation communities have shown considerable interest in such a toolkit. Developers could use this kit to create the AI in a game quickly without having to start from scratch. Ideally, the kit would consolidate the existing branches of work in the field and provide developers easy access to the fruits of mature research.

For the past couple of years, a small group of us at Stottler Henke Associates, an AI consulting company, have been working on techniques to write better AI behavior for simulations and games. We are focusing on a visual authoring tool that provides a way to quickly synthesize complex behavior, and are building a corresponding AI engine to run with a simulation or game. For game development uses, we see our tool as making the AI under-standable to game designers and end users, as well as improving developers' productivity. This work will also be useful for simulation developers and subsequently for analysts, operators, and instructors. Here, we explain some of our tool's features and how the AI engine processes the resulting content.

A new way to view behaviors

Instead of attempting to write a universal AI code base, we started by looking at what was common to most simulations and games. One requirement was that our software had to be accessible—more accessible than to just simulation or game developers. We knew that many game designers, while being perhaps the most creative part of a development team, did not know how to program a computer in the strict sense. However, we felt that if they could understand the visual representation of behavior, they could at least modify it, if not author behavior later with a developer.

About two years ago, we started with a graphical authoring tool based on one of our existing authoring tools for a Navy military simulation for tutoring student tactical-action officers. In the simulation, students command an Aegis cruiser and are confronted with realistic (and hostile) situations. The tool provides a visual way to look at behavior for a single entity but is not wedded to the actual simulation. The author can define his or her own AI vocabulary. We used *finite-state machines* to describe behavior. FSMs are common to almost all games and simulations, and we figured this would be a good place to start because of favorable properties, such as simplicity, compactness, and efficiency. After several iterations of extensions and reworkings of the visual interface, we arrived at a much more powerful way of articulating complex behavior.

However, the authoring tool only creates a description of behavior—it doesn't actually implement behavior. For that, we need a runtime engine that will take the description and make it operational in the game. The engine that we created had to satisfy three criteria: developers can easily interface to its API, it runs efficiently, and it is highly scaleable.

Our current software, BrainFrame, combines both the authoring tool and the runtime engine (see Figure 1). The AI author first uses the authoring tool to declare a basic vocabulary of actions and conditions. A primitive action could be to jump up or to compute a path from one location to another. A condition could be, "Is there a threat nearby?" The authoring tool only declares an entity's

1094-7167/02/$17.00 © 2002 IEEE

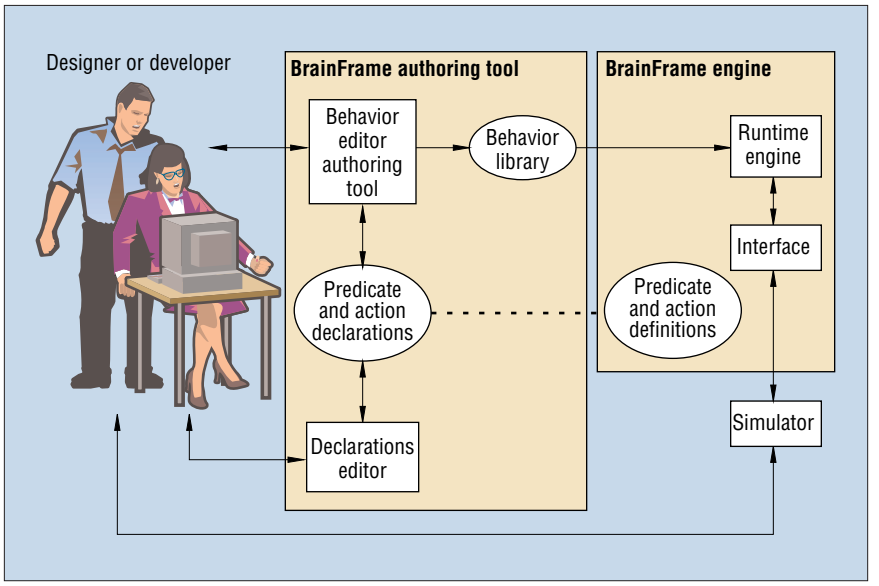

Figure 1. A diagram of the BrainFrame authoring tool and engine.

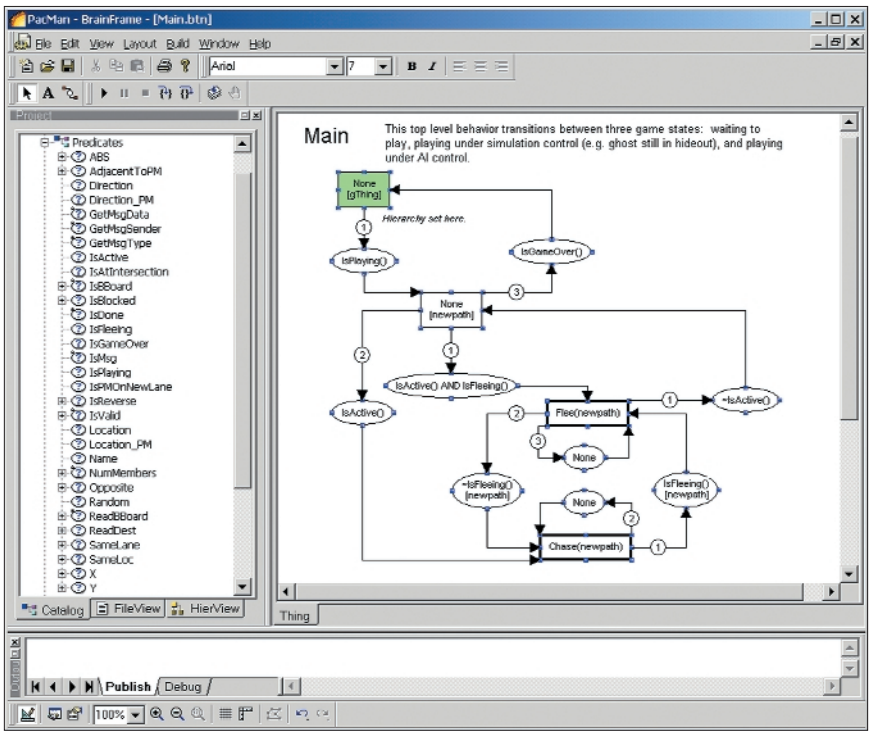

Figure 2. A screenshot of the BrainFrame editor.

capabilities for the behavior editor. For an entity to compute a path or detect whether a threat is nearby, it must rely on the BrainFrame engine.

BrainFrame uses the action and condition vocabulary as building blocks to construct FSMs, called *behaviors*. Each behavior consists of actions, conditions, and connectors. A behavior can invoke another behavior; thus, behaviors can be hierarchical and recursive. Altogether, the behaviors form the behavior library that the runtime engine uses. The runtime component then directs entities in a game. It does so indirectly through communication with an interface module between the engine and simulator. A developer writes computer code in this interface to operationalize the conditions and actions. A behavior doesn't need a computer code representation because it ultimately consists of simple actions and conditions. We've found in practice that as the types of information available to entities, and their capabilities, become better known, we will iteratively update the respective conditions and actions in the declarations editor and the interface.

The BrainFrame editor

Figure 2 shows a screenshot of the BrainFrame editor with a behavior built for a simple Pac-Man game used for demonstration purposes. The top-left pane holds a catalog of the condition and action vocabulary, variables, and a list of defined behaviors. The conditions appear under *predicates*. Whenever the user selects an item, its definition appears in the top-right pane. The lower pane is an output pane for behavior compilation and debugging.

Each behavior seen in the right pane of the editor consists of rectangles, directed lines, and ovals. Computationally, rectangles correspond to states in an FSM, while ovals correspond to conditions placed on state-to-state transitions. Each rectangle contains a reference to an action or behavior. References to behaviors appear as a bold, outlined rectangle. Anything appearing in parentheses is a parameter. Conditions are logical formulas that evaluate to true or false. Numbers on transitions determine the order of evaluation of conditions.

Three states are of special significance when interpreting a behavior at runtime. The *current state* denotes the action or behavior the entity is currently carrying out; a behavior can have exactly one current state at a time. The *initial state* is simply the rectangle in which the behavior starts. There can be only one initial state per behavior. When a *final state* is reached, we consider the behavior to have finished (in FSM parlance, it has reached an accepting state).

An action appearing in a rectangle will interact with the game engine through the interface module. For example, in a game, a human player might just press the R key to reload, while the interface would mimic pressing the same key for the artificial

entity. An action can also represent a deliberative or perceptual activity that has no direct physical effect on the game world, such as invoking a path-planning algorithm. References to behaviors in rectangles are handled completely within the engine. Ultimately, though, they boil down to primitive actions.

A behavior's current state changes according to *transitions*. A transition is a directed line connecting two states through zero or more conditions. Each transition has a set of conditions that, when all evaluate to true, will change the current state from the line's source to its destination.

The states and transitions are the core computational building blocks we use to articulate behavior. They are an extension of FSMs in that states can refer to other machines and that states and conditions can refer to operationalized code in the game interface.

Behaviors in action

As Figure 2 shows, a behavior has a Pac-Man ghost wait for a game to start, then alternates between chasing the Pac-Man and fleeing from it, until the game is over, at which point the cycle resumes. At the top is the initial node. It has no action but sets a local variable (the ghost's name). When the game starts (the condition `IsPlaying()` is true), the next node resets a path variable `newpath`. From there, the ghost stays in this state until the game world has the ghost outside its pen, free to move about (`IsActive()` is true). There are two possible transitions to either a chasing behavior or a fleeing behavior. Let's say it's the chasing behavior. The ghost stays in this state, continually recomputing paths toward the Pac-Man. As Figure 3 shows, the F and T markers denote forward and trailing intersections around the Pac-Man that are potential destinations for a ghost. The lighter lines are paths. Depending on the ghost's name, the path planner will select a forward or trailing destination. Should the Pac-Man eat a large "power pill," then each ghost will transition to the fleeing behavior. Should a ghost be eaten, the game takes over and returns the ghost to its pen in the center of the screen.

The BrainFrame AI engine

To better understand how the BrainFrame's AI engine processes behaviors, such as whether the ghost flees or chases, it is helpful to remember that behaviors can be hierarchical.

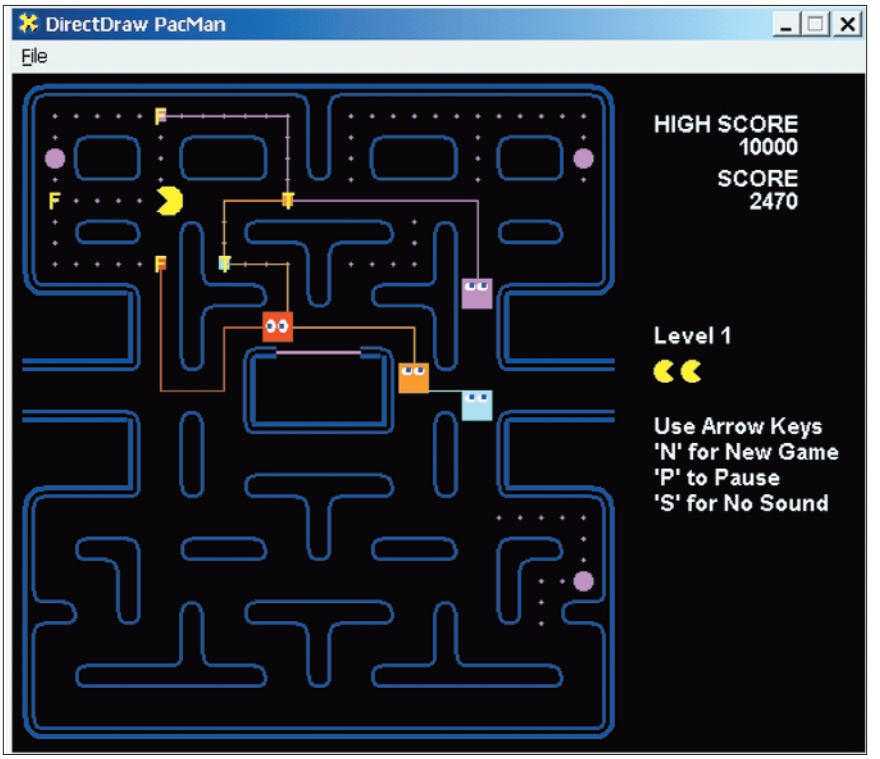

Figure 3. F's and T's show the potential forward or trailing paths of the ghosts depending on various game states.

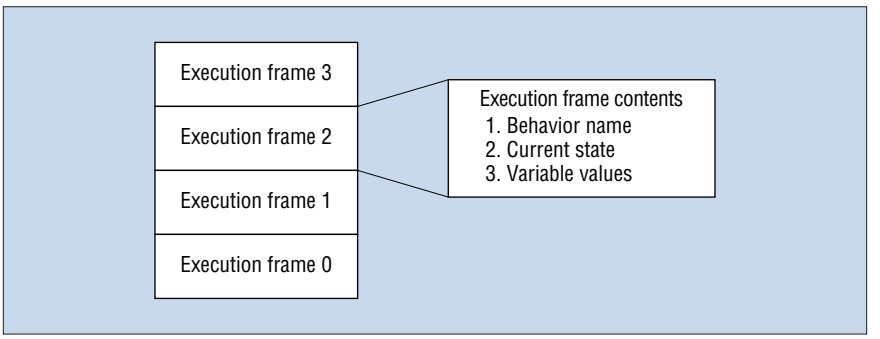

Figure 4. The execution stack containing frames of behaviors.

Any rectangle in the editor can refer to another behavior. When such a state becomes current, execution passes to that corresponding behavior's initial state. Each time this occurs, an *execution frame* is pushed onto the *execution stack*. The execution frame holds a single behavior's state. This includes the behavior's name, its current state, and any variable values. The execution stack maintains the set of frames; thus, it contains an entity's entire execution state. Figure 4 shows such a stack with frames.

AI engine algorithm

The AI engine processes these stacks using a two-step iterative algorithm. The first step is to execute the current action at the top-most frame.

The second step is to determine the next action. To do so, the engine examines the transitions coming out of each frame's current node, starting at the bottom because higher-level behaviors take precedence. In this way, if an entity is traveling to a location but notices a threat, it can handle it

immediately by discarding the frames having to do with traveling and by processing a frame to handle the threat. When a transition has a true condition, the engine discards the frames above and determines the next node containing a primitive action. This determination can cause more frames to be added on while the current node in the topmost frame is a behavior. Adding frames stops when the current node is an action—not a behavior.

Engine extensions

On the basis of our tests with the popular simulation games Half-Life and Civilization 2, we made four major extensions to increase the efficacy and expressiveness of the BrainFrame visual language:

- Global and local variables
- Interrupt transitions
- Shared blackboards
- Polymorphic indexing of behaviors

Global and local variables simply store values. They can be assigned values from arbitrary formulas, actions, or conditions. Global variables are available to all behaviors, while local variables can be used only in a single behavior. Interrupt transitions temporarily push on a special execution frame that takes precedence over frames below it.

Frequently, an entity will need to do some actions not associated with its primary task. For example, suppose an entity needs to notify its teammates of its location every 10 seconds. This need exists no matter what its current task—be it combat, navigation, spawning, or whatever. However, sending a message to teammates shouldn't derail the current task by forcing a transition to the BroadcastLocation action. A better solution is to create an interrupt transition that will temporarily divert the flow of control toward an essential behavior. When that behavior is finished, the flow reverts back.

A shared blackboard is used for broadcasting location: a way for entities to communicate to each other for an AI purpose. For multiplayer games, we found that some form of communication is necessary, especially for command-and-control situations. We implemented a protocol by which entities can publish and subscribe to information via a virtual blackboard. Even though we could have implemented this functionality separately in the game's interface, we felt the need was frequent enough to warrant an implementation within the engine.

The last major extension we made was *polymorphism*, an object-oriented programming term meaning that a single object can be interpreted differently depending on the situation. For BrainFrame, this translates into a single behavior having more than one definition. Which definition is invoked depends on the context. For example, in Pac-Man we have four ghosts, each with its own personality: When they pick a destination toward the Pac-Man, two will predict its future location, while the other two will move to its past location, thus closing off avenues for escape. In this case, the PickDestination behavior has two definitions, indexed by the ghosts' names. This indexing makes the behavior conceptually easier to understand.

Lessons learned

While working on BrainFrame, we learned that using hierarchical organization lets us modularize behavior and ends up simplifying the process of creating behavior. Naturally, for more complicated AIs, we ended up reusing several behaviors. The addition of polymorphism for behavior indexing also let us simplify our graphs, making them smaller and more understandable.

We also learned to never underestimate the vocabulary. It's easy enough to declare an action and a condition vocabulary, but implementing them on the game side is difficult. Once we crawled through the creation of an initial interface, sprinting toward advanced behavior was much simpler by comparison. Subsequent modifications to behavior are easy; a user can change the behavior definitions using the editor, recompile, and run the game. At this stage, the AI exhibited by game entities is limited only by the individual's imagination.

Finally, using the engine forces a clean separation between the AI and game engines. Or, as we've found working on existing games, our prescribed interface forced us to clean up the existing code.

Our next step is to incorporate the BrainFrame toolkit into a software tool that will let military planners rapidly create complex customized war game simulations without the assistance of a programmer. This tool will provide a collection of visual editors allowing the user to specify simulations' various elements and their interactions; the user will define entity behaviors in the simulation using the BrainFrame editor. The runtime engine will also serve as the underlying simulation architecture for this new tool, executing not only the user-defined entity behaviors but also the customized rules of the simulation itself that the system automatically generates. By using tools such as these, simulation and game developers can speed their development time while empowering designers and end users to build on their work. ■

Acknowledgements

Our research is supported in part by Air Force Research Laboratory grant F30602-00-C-0036.

Daniel Fu is a project researcher at Stottler Henke Associates. He has a BS in computer science from Cornell University and a PhD in computer science from the University of Chicago. His research interests include autonomous agents, planning, and simulation. Contact him at fu@shai.com.

Ryan Houlette is a lead software engineer and project manager at Stottler Henke Associates. His research interests include intelligent interfaces, autonomous agents, and automatic generation of simulations and interactive worlds. He holds a BA in computer science from DePauw University and an MS in computer science (AI specialization) from Stanford University. Contact him at houlette@shai.com.