Towards an AI Behavior Toolkit for Games

Ryan Houlette[*], Daniel Fu[*], and David Ross[**]

[*]Stottler Henke Associates Inc
1660 S Amphlett Blvd, Suite 350
San Mateo, CA 94402
[**]Air Force Research Laboratory
525 Brooks Road
Rome, NY 13441-4505
{houlette,fu}@shai.com, rossd@rl.af.mil

Abstract

We discuss our work to create an AI engine and editing toolkit, suitable for both AI researchers and game developers. We describe the run-time engine and its interface, as well as a simple behavior editor that allows non-programmers to author behaviors. Initial work was conducted using the Half-Life SDK.[1]

Introduction

To date there exists a disparity between AI research and game development technologies (Laird and van Lent 2000). We are currently creating an AI behavior run-time engine and editor that both communities could potentially use. This effort comes out of a growing interest in the gap between AI as done in laboratories and AI as done in games. AI researchers want to determine how their AI can work in games, and if not, why. Game developers want to know how AI research can help produce better game AI. Coming out of the AI community, we believe a common development kit can bridge the gap by providing ways for AI researchers to test their algorithms in existing games, and for game developers to have access to state-of-the-art AI results.

Such a kit has obvious benefits for both communities. On the game development side, the kit will provide a straightforward method of supporting games. This allows developers to build custom extensions based on well-understood standards. As well, mod communities will have support for AI authoring, instead of coping with an undocumented body of code. Reaching even further, visual-oriented editors make the AI behavior accessible to non-technical people such as game designers. Game designers, while not necessarily technically oriented, often contribute the lion's share of the innovation in games.

[1] This research is supported by Air Force Research Laboratory grant F30602-00-C-0036.

On the research side, the kit will offer a game interface so that AI researchers don't have to build their own. Besides saving time and effort, having a common interface can foster the creation of benchmarks. So far, AI in a game is more art than science, as it can be difficult to tell when a good AI research "result" has been obtained. The AI field doesn't have an objective set of criteria—they are borrowed from other fields such as math, philosophy, or engineering—and it is questionable sometimes which criteria are most appropriate. A benchmark would enable researchers to measure and report the performance of algorithms in a more objective manner through a common interface and framework. Finally, the kit will focus AI research on a set of core game AI problems. Having a developer kit means having common ground for both communities.

Overview of Toolkit

For the development of the engine and editor, we've chosen Half-Life, a popular game in the first-person-shooter genre, as our initial testbed. Half-Life offers a challenging domain in which to create realistic and interesting artificially-intelligent characters, emphasizing real-time response, navigation in a 3D world, and sophisticated multiplayer coordination among entities. We are working with the freely-available Half-Life SDK, released by Valve Software, which provides access to the game engine and allows modification or even complete replacement of the existing artificial intelligence code.

The toolkit consists of two main components: a core AI run-time engine that interfaces directly with the game, and a visual editor, called BUTTON, used to build behavior transition networks for game characters. The AI run-time engine is designed to have a highly modular architecture allowing for selective, plug-and-play inclusion of different feature sets and capabilities, such as path planning, learning, communication, and emotion. At core of the engine is a flexible, efficient execution model based on specialized directed graphs called *behavioral transition networks*. These networks have the virtue of being easy to under-

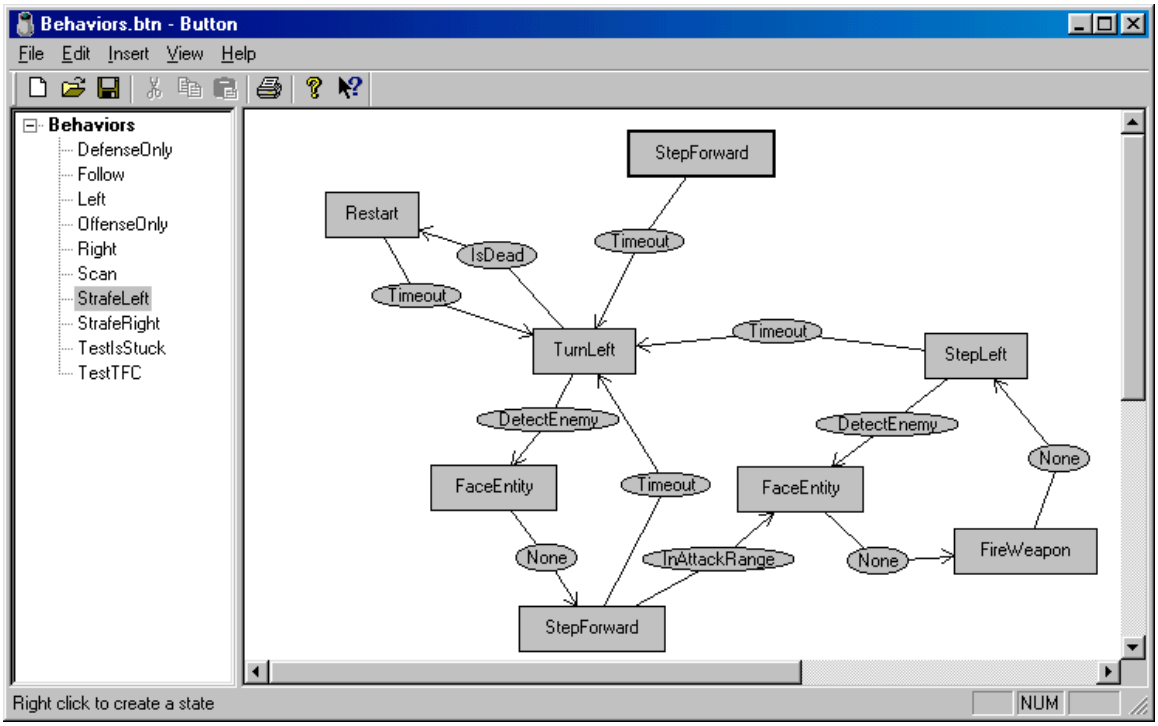

Figure 1: Screenshot of the behavior editor showing the "StrafeLeft" behavior.

stand, code, build, and test, and also boast compact code size and fast execution.

Visual Editor

The visual editor is a stand-alone application that allows the game developer to graphically specify the behavior of characters in the game. The intent is to minimize the amount of programming ability required to use the editor so that people with non-technical backgrounds can use it with a minimum of training to create complex and realistic behaviors. We envision the editor as a visual programming environment, where the author can drag and drop behavior components onto a canvas and connect them together with various kinds of triggers, links, and conditions. Once the author has finished editing a set of behaviors, the editor then compiles the visual specifications into low-level binary code that can be directly executed by the run-time engine for maximum speed.

Our prototype version of the behavior editor is based upon an existing SHAI product called the Tactical Action Officer Intelligent Tutoring System Scenario Generator. This software is an authoring tool that allows the user to create finite state machines specifying the behavior of a simulation entity. We have modified it so that we can author simple behaviors for autonomous "bots" in Half-Life. Figure 1 is a screenshot showing the editor. The author can define behaviors by creating states and transitions in the right pane. The editor lists all behaviors in the left pane. In this case the left pane shows the "StrafeLeft" behavior and the right pane shows the corresponding definition.

Run-time Architecture

In our run-time engine, each AI-directed entity associates with one or more behaviors that dictate how it will act in the game world. Behaviors are represented as hierarchies of transition networks (also called behavioral transition networks, or BTNs). These hierarchies consist of two types of elements, *transitions* and *nodes*.

A node in a BTN represents an action that the entity may possibly perform at some point during the game. Two nodes in a BTN are of special significance. The *current node* of a BTN denotes the action currently being carried out by the associated entity; a given BTN may have exactly one current node at a time per execution frame. The *initial node* of a BTN is simply the node with which the BTN begins execution (i.e., the BTN's initial current node).

Note that the actions represented by a node may be either concrete and primitive – for example, MoveTo(x,y,z) or DoNothing(t) – or more abstract and complex – for example, FindNearestEnemy. An action may also represent a deliberative or perceptual activity that has no direct physical effect on the game world. Primitive actions tend to directly interact with the game engine through API calls,

while complex actions are generally carried out by sub-BTNs or specialized behavioral modules.

A transition in a BTN is a directed arc connecting two nodes X and Y (or looping from one node back to itself) and indicating a potential direction of control flow. A decision process – typically a set of logical conditions, but possibly something more sophisticated – is associated with each transition. A transition is said to be *active* if its decision process returns a "true" result; an active transition indicates that the BTN may change its current node from node X to node Y.

As stated above, BTNs may be hierarchical – that is, any node in a BTN may link to another arbitrary BTN or set of BTNs (either enumerated explicitly or specified by a list of descriptors). When a node with such links becomes current, execution passes to one of the linked BTNs, which begins at its initial node. Each time this occurs, a new *execution frame* is put on the *execution stack*, which maintains the behavioral state at each level of the currently-executing BTN hierarchy for a given entity. This tiered structure allows for natural decomposition of tasks and abstraction of high-level behaviors.

At each time step during execution, the engine examines the entity's execution stack to determine the next transition to be followed. Transitions are evaluated for the current node in each execution frame, starting at the bottom of the stack and stopping when an active transition is found. That transition is then taken, changing the current node and causing the associated action to be performed by the entity. In addition, all execution frames above the one with the active transition are popped from the stack.

Example of Execution Flow

To give an idea of how execution actually works, here is a simple example. It refers to Figure 2, which shows the current execution state for a single entity. The rectangles (with rounded corners) denote nodes, and the arrows between them represent transitions. Nested nodes indicate different levels of the hierarchy, and nodes with dashed borders are current nodes (in their particular execution frame).

At the current moment, the execution stack for this entity looks like this:

Execution Frame	Current Node	Possible Transitions
3	U	U→V, U→W
2	Q	Q→R, Q→S
1	L	L→M, L→N

Thus, when it becomes this entity's turn in the execution cycle, the first thing that will happen is that the action associated with node U will be called, since U is the current node in the topmost execution frame.

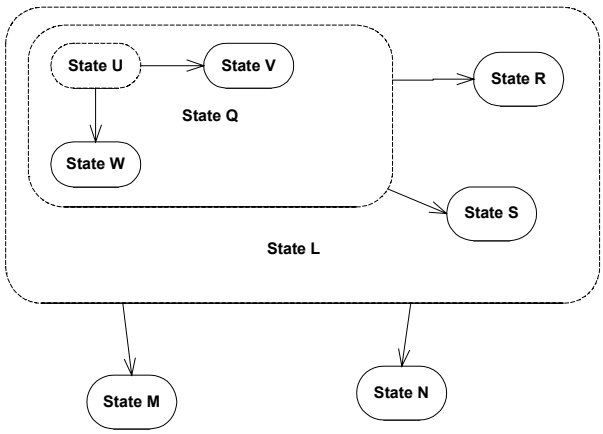

Figure 2: Current execution state for a single entity.

Next, all viable transitions will be evaluated, starting with the bottom of the stack and working up. Evaluating transitions in this order allows more general, ongoing actions (i.e., higher-level plans) to take precedence over low-level, short-term actions.

Thus, the possible transitions for execution frame #1 are checked first:

T_{LM}->Evaluate() == false
T_{LN}->Evaluate() == false

Suppose that neither of these is satisfied. Evaluation now moves to execution frame #2:

T_{QR}->Evaluate() is true
T_{QS}->Evaluate() is false

Note that even though T_{QR} was determined to be satisfied, T_{QS} was still checked. The decision to follow a transition is not taken until all transitions out of the current node in the execution frame under consideration have been evaluated.

In this case, T_{QR} is the only active transition, so there is no choice to be made. T_{QR} is traversed, which pops off the now-irrelevant execution frame #3, sets the current execution frame to be #2, and sets the current node in that frame to be R.

At the end of the execution cycle for this entity, the execution stack for this entity looks like:

Execution Frame	Current Node	Possible Transitions
2	R	not shown
1	L	L→M, L→N

Discussion

The development of a robust, flexible, and yet comprehensible interface between the core game engine and the AI engine is key to the creation of a useful toolkit. It is also perhaps the most difficult component of the project. There are two main reasons for this. First, specifying a usable and powerful API is, in general, a hard problem, because as the power and flexibility of the API increases, so too does the number of abstractions and conceptual layers that must be introduced, and this growing complexity tends to increase the learning curve and discourage new users.

Second, most AI approaches are, by their very nature, knowledge-intensive, and the internal knowledge representations used by games – which are, almost without exception, custom solutions developed for that specific game – vary drastically, which means that any AI toolkit hoping to be general must be extremely versatile in order to allow the AI component adequate access to the game information it needs. There is a delicate balance to be struck here: if the assumptions made by the toolkit are too broad, then the gap between any particular game and the toolkit API will be large, requiring an impractical amount of custom code to be bridged. If the interface is too focused, on the other hand, the toolkit's applicability will be limited.

In order to combat these difficulties, we have chosen an iterative development plan that will allow us to incrementally develop our interface and architecture, refining our design as we learn more about the demands and necessities involved through experimentation. As mentioned above, our initial development has been with the Half-Life SDK; as the project progresses, we intend to add other testbeds, possibly including Unreal Tournament, as well as a turn-based strategy game like Civilization. The use of multiple testbeds will help ensure that our toolkit is general enough to be used with a variety of games.

We are currently beginning our second pass through the development cycle, having already completed a prototype version of both the runtime engine and the behavior editor. We have nearly finished a revised design document based on the lessons learned from this prototype, and we expect to begin implementation of the first release version of the toolkit soon.

Related Work

The notion of having a visual representation of AI behavior is not new, both in academia and in industry. MacKenzie, Arkin, and Cameron (1997) describe the MISSIONLAB system that allows an author to specify the behavior of multiple robots. The author does this visually using similar hierarchical state and transition links. KHOROS (www.khoral.com) is a popular visual editor for image processing that allows users to string together operators into a flow diagram. Each operator comes from a standard library, or is defined by the user using standard C code. In industry, there are few visual editors for games. Perhaps most notable is the Motivate package from the Motion Factory (www.motion-factory.com). Its use is limited to development companies and is not freely available for research use.

Conclusion

We have presented an overview of an ongoing project to develop an AI software toolkit for computer games. The intent of this kit is twofold: to "raise the bar" on the quality and sophistication of AI used in games by providing powerful, easy-to-use tools for putting AI into games; and to help legitimize game AI as a reputable branch of artificial intelligence by providing a standard interface and framework in which researchers can implement and experiment with their algorithms, techniques, and approaches across a variety of game testbeds.

References

Laird, J., and van Lent, M. 2000. Human-level AI's Killer Application: Interactive Computer Games. In *Proceedings of the National Conference on Artificial Intelligence*.

MacKenzie, D., Arkin, R.C., and Cameron, J., 1997. Multiagent Mission Specification and Execution. *Autonomous Robots*, 4(1):29-57.