7

Real-World Behavior Trees in Script

Michael Dawe

7.1 Introduction

While there are many different architectures for an AI programmer to pick from, behavior trees are one of the most popular algorithms for implementing NPC action selection in games due to their simplicity to code and use. They are quick to implement from scratch and can be extended to add additional features or provide game-specific functionality as needed. While not as simple as a finite-state machine, they are still simple enough to be easily debugged and designed by other team members as well, making them appropriate to use on games with a large team implementing the behaviors.

On *Kingdoms of Amalur: Reckoning*, we wrote a behavior tree system that used behaviors written in script, while the algorithm itself was processed within the C++ engine. This allowed the design team to have rapid iteration on the behaviors while the programming team retained control over the algorithm features and how the engine processed the behavior tree. Here, we present a functional implementation of a simplified version of the algorithm (available on the book's website http://www.gameaipro.com). It can be used "as is" or extended for a more demanding application. Additionally, we discuss some of the pros and cons we found using such a system in the development of *Reckoning*.

7.2 Architecture Overview

The behavior tree algorithm implemented in the sample code is straightforward and assumes only the knowledge of a tree data structure. Each node in the tree is a *behavior*,

Listing 7.1. Pseudocode for running the behavior tree. Called with the root node to start, this recursively determines the correct branch behaviors to run.

```
process_behavior_node(node)
    if (node.precondition returns true) {
        node.action()
        if (node.child exists)
            process_behavior_node(node.child)
    } else {
        if (node.sibling exists)
            process_behavior_node(node.sibling)
    }
```

with some *precondition* defining when that behavior should run and an *action* defining what the agent should do to perform that behavior. A root behavior is defined to start at, with child behaviors listed in order. Starting from the root, the precondition of the behavior is examined, and if it determines that the behavior should run, the action is performed. The algorithm would continue with any children of that behavior. If a precondition determines that a behavior should not be run, the next sibling at that behavior is examined in turn. In this manner, the algorithm recurses down the tree until the last leaf behavior is run. The pseudocode for the algorithm is shown in Listing 7.1. This process on the behavior tree can be run as often as needed to provide behavior fidelity, either every frame or considerably less often for lower level-of-detail.

In the code sample, we define classes for each `Behavior`, as well as a `BehaviorMgr` to keep track of each `Behavior` loaded. The `BehaviorTree` is its own class as well, so that we can define multiple trees that use the same behaviors and process them separately. Since each `BehaviorTree` is made up of a list of `Behaviors`, multiple characters running the same behavior tree can run separate instances of the `BehaviorTree` class.

7.3 Defining Script Behaviors

There are several compelling reasons to define behaviors in script. First, when developing intelligent agents, quick iteration is often a key factor in determining final behavior quality. Having an environment where behaviors can be written, changed, and reloaded without restarting the game is highly desirable. Another reason to use a scripting language might be to take advantage of your team. On *Reckoning*, a large percentage of the design team had a programming background, making it feasible for them to implement behaviors in script without much programming input or oversight. While some programming time was needed to support the creation of new ways to pass information back and forth between the C++ engine and script, overall the time spent by engineers on the behavior creation process was much less than it would have been otherwise.

In order to write our behavior tree to take advantage of behaviors written in script, we first need to integrate a scripting language to a C++ engine. For *Reckoning*, we chose Lua, a popular scripting language within the games industry. Since Lua is written in C, it can

easily be plugged into an existing engine, and since its source is distributed for free, it can even be modified as necessary and compiled into the game.

Lua defines a native data structure—a *table*—which is analogous to a *dictionary* or *map* in other languages. For each behavior, we defined a table with that behavior's name (to avoid name collisions). The members of the table were functions named "precondition" and "behavior." With known names, the C++ algorithm could look for and call the appropriate functions at the correct times.

7.4 Code Example

Besides the behavior classes, the code sample used in this article also defines a `LuaWrapper` class to manage the Lua integration, and an `NTreeNode` class as a generic tree class. In `main()`, a `LuaWrapper` is created and used to load all *.lua files in the Scripts directory, where every *.lua file is a well-defined behavior. While the `LuaWrapper::load_all_scripts()` function is written for Windows systems, all operating-system specific calls are in that function, so porting to a different system should be confined to that function.

From there, a test behavior tree is created based on the script files loaded. Using the `add_behavior_as_child_of()` function, an entire behavior tree can be created from scratch. Finally, the tree is run using `process()`, which simply starts the recursive function at the root of the behavior tree and tests each behavior in turn.

7.5 Integration into a Game Engine

While functional, the sample provided holds more power if plugged into a full game engine. The `Behavior` and `BehaviorTree` classes can be taken "as is" and extended as needed. The `LuaWrapper` could also be taken as written, but ideally an engine would include functionality for reloading the Lua state at runtime, in order to take advantage of being able to rewrite behaviors and test them without restarting or recompiling the game.

Each agent can define its own `BehaviorTree`, either in code or as some sort of data file. For *Reckoning*, behavior trees were assets just as behaviors were, so trees had their own manager and data definition for ease of sharing among multiple different types of NPCs. If the behaviors are written generically enough, many different agents could share not only the behaviors, but even whole trees.

7.6 Script Concerns

While the benefits of having script-defined behaviors are manifest, there are particular concerns that should be kept in mind if using the approach.

Perhaps the first thing to come to mind for game programmers is performance. While written in C, Lua is still a garbage-collected language, and has a floating-point representation for all numbers within the Lua environment. With this in mind, *Reckoning* took a few precautions to safeguard the framerate of the game from poor script performance.

First, since Lua can be compiled directly into the game engine, all Lua allocations can be routed through whatever allocation scheme the engine implements, which means it's possible to take advantage of small-block allocators to avoid general fragmentation issues. By preemptively garbage collecting at known times, it's possible to prevent the Lua garbage

collector from running anytime it would be disadvantageous. In particular, *Reckoning* ran the garbage collector every frame at a predetermined time to avoid mid-frame collection that can occur when left up to Lua.

To further increase performance when in Lua, the source was changed to make Lua's internal number system use traditional integers instead of floating-point numbers. This had a few consequences for scripters, the most obvious of which was dealing with integers for percentages, i.e., the number "56" instead of "0.56" for 56%. Once this was well communicated, it was merely a matter of scripting style.

Trigonometry and geometry became impossible to complete within Lua, though, and while this is precisely the outcome planned for, it was a larger workflow change. Since the point was to avoid any complex math in script, it was planned that anytime a behavior or other script needed a trigonometric or geometric problem solved, it would ask the C++ engine for an answer. This meant that while most of the mathematical calculation was kept out of script, more programmer time was required to write and test the necessary functions for script any time a new result was needed.

In general, though it was still a positive time gain to have designers writing more script behaviors, programmers could not be entirely hands-off during behavior development. After *Reckoning* completed, both the designers and programmers agreed that more formal engineering oversight was needed in the scripting process; so while a majority of behaviors in the game were written by designers, the team thought more collaboration would be warranted. A suggested workflow was to have members of the engineering team code review script check-ins, though periodic reviews would also work.

7.7 Enhancements

While the sample can be used "as is," part of the appeal of a behavior tree is implementing extensions as needed for your own project. There are a wide variety of additional features that can be added to this base tree. Here are some examples used on *Reckoning*.

7.7.1 Behavior Asset Management

Although this example just loads files in a given directory, if the behavior tree system needs to interact with a large number of behaviors, it will become easier to have some sort of behavior asset manager to load and track each behavior. While initially this simply shifts the responsibility of loading the behaviors to the new asset manager, the manager can add new functionality by creating a unique ID for each behavior. Trees can reference behaviors by this ID, while the manager can enforce uniqueness of names among behaviors. By having a centralized place to do error-checking on the scripts, finding and recovering from data errors can be handled more easily.

Having an asset manager for your behaviors has other advantages, as well. While this sample creates the trees as part of the program, ideally trees are defined by some data file that's read in when the game starts. This allows the development of external tools to create trees or even define trees containing other trees.

7.7.2 Behavior Definition Extras

As noted, our behavior definitions are simply tables in Lua, which are analogous to dictionaries or maps. Tables can hold arbitrary data, a fact this implementation takes

advantage of by storing functions within our behavior table. Since there's no limit on the data, though, we can also store any data with the behavior we want besides just the precondition and behavior functions themselves. For example, a behavior could store a separate table of parameterized data for use within the behavior, or data for use by the behavior tree. In *Reckoning*, behaviors specified a hint to the behavior-level-of-detail system based on how important it was that they run again soon. For example, creatures running combat behaviors had a higher probability of getting to process their tree than creatures merely walking to a destination.

7.7.3 Behavior Class Improvements

The `Behavior` class itself can be improved to allow for faster processing of the tree. In this example, behaviors must define a precondition function and a behavior function, or else the tree will fail to process correctly. It is possible to use the `lua_isfunction()` family of functions to determine if the behavior or precondition functions exist before calling them. While the behavior tree could push the known function location onto the Lua stack to determine its existence every frame, a better solution is one where the behavior itself checks and tracks what functions are defined when it first loads. Then the behavior tree can call or skip a behavior function call based on a flag within the behavior itself without incurring a significant performance cost while processing the tree.

7.7.4 Previously Running Behaviors

Often it is useful when debugging a behavior tree to know which behaviors were running on a given frame. The behavior tree can keep track of which behaviors were running on the last frame or on an arbitrary number of frames prior. The smallest way to do this is by using a bit field. By taking advantage of the fact that a behavior tree is laid out in the same way every run, we can assign the first position in the bit field to the root, then the next its first child, followed by any children that child behavior has before moving on similarly. Algorithmically, we can then simply mark the behaviors to be run while checking behavior preconditions, then save that bit field off when we are finished processing.

In fact, a behavior tree can be compressed considerably using this technique. For example, instead of storing the behaviors in a tree, once the behaviors are loaded by an asset system and given a unique id, the `BehaviorTree` class can store an array of pointers to the behaviors, and the tree can store indices into that array, which simplifies the bit field approach.

7.7.5 on_enter/on_exit Behavior

Once a list of previously running behaviors is established, a behavior can define a function for the first-time setup or a cleanup function for when it ceases running. As a part of *Reckoning*, we defined `on_enter` and `on_exit` functions for each behavior. To implement these, the behavior tree class needs to track over subsequent `process()` calls which behaviors were running the previous time, as above. If a list of behaviors run the previous tick is kept, then any behavior in the previous list but not in the current one can call its `on_exit` function before new behaviors are started. `On_enter` and `behavior` functions are then called in order.

7.7.6 Additional Selectors

The algorithm can also be extended by changing the way behaviors are selected. Some different selectors used on *Reckoning* included *nonexclusive*, *sequential*, and *stimulus* behaviors. Each of these slightly changed how behaviors could be selected to run or altered the logic of how the tree progressed after running a behavior.

Nonexclusive behaviors run as normal behaviors do, but the tree continues checking siblings at that tree level after doing so. For example, a nonexclusive behavior might play a sound or set up some knowledge tracking while leaving it to other sibling behaviors to determine an actual action.

Sequential behaviors run each of their children in order so long as their precondition returned true. An example might be a behavior to perform a melee attack, with children to approach the target, launch the attack, and then back away. So long as the parent melee behavior returns true, the tree will execute the child behaviors in order.

Stimulus behaviors are a way of hooking the behavior tree up to an in-game event system so that agents can define reaction behaviors to events happening around them. Each stimulus behavior defines a particular stimulus, such as spotting the player or hearing a sound, which it can react to. Stimulus behaviors are a way of specializing a commonly used precondition. In *Reckoning*'s implementation, stimulus behaviors were treated exactly as normal behaviors with a separate precondition function that would check for a defined stimulus on the character running the tree. This specialized precondition also handled cleanup of the stimulus when finished.

Any kind of selection algorithm can work with a behavior tree, which is one of the major strengths of the system. For example, a utility-based selector could pick among its children based on their utility scores, or a goal system could be implemented that picks children based on their postconditions. While any selection algorithm can be made to work, often they will change how the behaviors must be defined, either through additional functions or data needed by the selector. The flexibility of using any kind of selector must be weighed carefully against the time and cost of implementing each different algorithm.

7.8 Conclusion

Behavior trees are a flexible, powerful structure to base an agent's decision-making process around, and utilizing script is one method to drive faster iteration and greater ease of behavior authoring. Being able to edit and reload behaviors at runtime is a huge advantage when refining and debugging behaviors, and having a data-driven approach to behavior creation opens up the process to a much wider group of people on the team, helping production speeds. Additionally, with the behavior tree algorithm being as flexible as it is, improvements and game-specific features can be implemented quickly, and with behaviors implemented in script, each can be updated to take advantage of the new features quickly. Changing their parameters can be done without recompiling or restarting the game, so rapid testing of these features can be accomplished.

If a behavior tree is a fit for a game, having script support for implementing the behaviors provides tremendous flexibility. Though careful analysis of the performance costs is necessary any time a scripting language is used, strategies can be employed to minimize the impact while maintaining the advantages of having a rapid iteration environment for behavior development.