Modular Behavior Trees: Language for Fast AI in Open-World Video Games

Tomáš Plch [1,2] and **Matěj Marko** [1] and **Petr Ondráček** [1]
Martin Černý [2] and **Jakub Gemrot** [2] and **Cyril Brom** [2]

Abstract. As the graphical representation of computer games gradually becomes comparable to cinema, previously neglected non-graphical game aspects start to play an important role in maintaining believability of the game world. One such aspect is the behavior of non-player characters (NPCs), which should appear intelligent and purposeful in the ideal case or not completely stupid at the very least. A good and fast language to express the behaviors is vital for success. We present a visual agent language inspired by behavior trees that was approved for deployment at the core of AI system for an upcoming high-budget open world game.

1 INTRODUCTION

Computer games are a very specific AI application area. A particularly interesting subclass of games are those featuring a large 3D world that is open (the player may roam freely through the environment) and inhabited by a plenty of non-player characters (NPCs).

The NPC AI of a typical game may be divided into several main components, although some of the components may be diminished or absent. *Combat AI* is often the largest AI subsystem. It may be further divided into *enemy AI* that guides NPCs opposing the player and *ally AI* that controls NPCs trying to help the player in a fight.

Non-combat AI governs the rest of the NPC behavior. It may be further divided into *direct interactions* with the player (e.g., dialogues, barter, ...) and *ambient AI* which covers the daily life of the NPCs and other actions they perform on their own. It is the ambient AI that makes the world appear alive to the player.

Enemy AI and direct interactions with the player are well managed in contemporary games. The ally AI is more problematic, as the NPC is required to be helpful to the player without being able to see "into his head". Nevertheless, multiple games have tackled this issue with results that were workable, if not satisfactory.

Ambient AI appears to be the least developed of the aforementioned components. Contemporary games have very limited support for ambient AI — in particular, almost all high-budget commercial games do not actually simulate NPC behaviors outside the area directly surrounding the current player's location.

This is not so surprising when we consider that the time available for AI is severely limited: our system was required to spend at most 5ms per game frame using a single processor core (the game should run at least 30 frames per second). Although handling hundreds of NPCs in this time frame is possible on modern hardware, the NPC deliberation must be very quick and therefore must use some kind of reactive planning; more complex reasoning is not acceptable.

In this paper we present an agent-based language designed to handle NPC AI easily and efficiently. The language also supports level of detail (LOD) AI ([1]) to further decrease computational cost.

[1] Prague Game Studios, Czech Republic, email: tomas.plch@gmail.com, MattEntrichel@gmail.com, petr.ondracek@warhorsestudios.cz
[2] Charles University in Prague, Czech Republic email: {cerny.m,jakub.gemrot}@gmail.com, brom@ksvi.mff.cuni.cz

2 MODULAR BEHAVIOR TREES

Reactive planning has been at the core of several agent-based languages, notably POSH [2] which composes complex behaviors hierarchically from simple primitives in a tree-like structure. A similar approach with slightly different semantics, called behavior trees [4] has gained popularity in the game AI community and has become a de facto industry standard.

In plain BTs, evaluation of a node of the tree may return three possible values: `success`, `failure` and `running`. The leaves are atomic actions and conditions, they return `success` when the action is done or condition is fulfilled, `failure` if the action fails or condition is violated and `running` if more is to be done. The internal nodes (called *composites*) are either *selectors* or *sequences*; selectors return `success` when the first child node succeeds and do not evaluate the rest of the children. They return `failure` only after all children fail. Sequences need all of their children to succeed in order to return `success` and fail with first failing child. Both return `running` if the evaluated child returns `running`.

Further extensions to the behavior tree (BT) formalism including *decorator* nodes altering the execution context of the subtree [5] and parallel execution [3] were proposed.

We have however identified downsides to the basic idea and improved both the syntax and the semantics of BTs to better express complex behaviors. We call the augmented language *modular behavior trees* (MBTs). The key issues we addressed were the execution model, variable support, missing synchronization and communication mechanisms, no explicit time awareness and tool support.

2.1 Node execution

In a plain BT, the whole execution is stateless and the conditions that guard the individual tasks are continually reevaluated. This introduces high reactivity to external stimuli, but it may be computationally intensive. In some other implementations, the composites have internal state and continue evaluating their children starting at the first one that returned `running` in the previous iteration. However, reactivity may be greatly reduced this way. We have decided to increase flexibility and let the individual nodes control what state should be kept and how children are evaluated.

In stateless BT variants, there is no guarantee that an action node that is running in the current update will be evaluated in the next update. Thus a newly started action often finds the NPC in an intermediate state and external mechanisms are needed to keep the NPC's state consistent.

In MBTs, nodes that were running in the previous update are notified if they should stop executing due to an event higher in the tree and they are allowed to clean up. We implemented two ways to interrupt node execution: the node could either be *suspended* and retain its state, or *halted* and clear its state. However, some behaviors may not be interrupted abruptly without threatening consistency (e.g., a priest should drop the book he is reading before switching to a combat behavior). Thus MBTs allow for intermediate states that let the node execute until it suspends or halts completely.

Once again, the individual nodes are in full control of how updates and interrupt signals propagate to their children and how states of their children affect their own state. For example, we have created a decorator node that reacts to interrupts by evaluating a given cleanup subtree. This further increases designer control over the behaviors and eases maintaining behavioral consistency.

Among the other nodes we have created are stateful and stateless variants of selectors and sequences, loops, decorators for time-limited execution, parallel execution of multiple subtrees, "calls" of external subtrees and decorators that alter the result returned by a subtree. We also introduced support for creating finite state machines [6] inside the trees.

Since the composites have full control over execution, they may return control before an actual action is issued and continue consistently in the next update. This has let us to enforce upper bounds on time spent in MBT evaluation. The execution semantics also allow for good debugging support. In particular, breakpoints may be attached to various transitions of the node state or to individual updates.

2.2 Types and variables

In most engines, the data accessible from BT is limited to either a set of hard coded states or values provided by the engine (e.g. a boolean InDanger indicating a serious threat, or object reference FocusTarget representing the game entity the NPC is looking at) or information in a simple "key – value" pairing. In order to create more versatile data model we have introduced a simple type system.

Every type definition is similar to a struct construct of the C language. The individual members are either primitive types (boolean, integer, float, string) or types defined previously. The MBT than defines variables which may be substituted for any parameter of a node.

2.3 Synchronization and communication

We have introduced a powerful messaging system. Messages are simply data of a predefined type sent from one NPC to one or multiple other NPCs. An NPC has a list of associated *inboxes*, each inbox has a type of data it receives, priority and possibly further filtering logic.

The most common synchronization task in the game is the need for multiple NPCs to start a task at the same moment. We have introduced a special node that blocks at a semaphore until a specified number of NPCs subscribe to the semaphore. Our extended node execution model allows for consistent locking and releasing of the semaphore in response to changes of the node state. Thus the NPC may easily have a subtree executed in parallel to the synchronization node and perform meaningful actions while waiting.

2.4 Other improvements

Plain behavior trees are internally unaware of passage of time and handle time only at action level. Most notably, at most one action is performed during a single tree execution. This leads to more complicated design when updates to the tree are sparse due to LOD policy. MBTs are implicitly time-aware as the time delta from previous execution is given to the nodes. Each action consumes a part of the delta and the execution stops once the remaining delta is zero. This way varying time flow rate for different NPCs is possible.

We have created a visual editor for MBTs with drag and drop support and debugging interface (breakpoints, variable introspection).

3 EVALUATION

We have performed a preliminary quantitative evaluation of the speed of the whole system in two scenarios: In first scenario there were 200 NPCs. Each of them had a simple tree guiding the NPC to move to random locations. The tree updates took 1.29 ms on average (sd 3.09). The second scenario was a production one - 16 NPCs carrying out their daily routines (e.g., hoeing fields, visiting pub, eating). In this case, the tree updates took 0.29 ms on average (sd 2.86). As all the NPCs were updated every frame without any LOD optimization, proper use of LOD should be enough to keep the system below the 5ms per frame limit in the envisioned production load.

4 CONCLUSIONS

We have presented an agent-based language that is the core of an AI system for a large open world RPG game currently under development. The language has been shown to handle the intrinsic complexity of NPC behavior well without sacrificing performance. The system has been approved by project leads for deployment in the game and has replaced the system shipped with the game engine.

ACKNOWLEDGEMENTS

This research is partially supported by the Czech Science Foundation under the contract P103/10/1287 (GAČR), by student grants GA UK No. 559813/2013/A-INF/MFF and 655012/2012/A-INF/MFF and partially supported by SVV project number 260 104.

REFERENCES

[1] Cyril Brom, Ondřej Šerý, and Tomáš Poch, 'Simulation level of detail for virtual humans', in *Intelligent Virtual Agents*, pp. 1–14. Springer, (2007). LNCS 4722.

[2] J. Bryson, *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.*, Ph.D. dissertation, Massachusetts Institute of Technology, 2001.

[3] A. Champandard, 'Enabling concurrency in your behavior hierarchy', *AIGameDev.com*, (2007). `http://aigamedev.com/open/article/parallel/` Last checked 2014-08-10.

[4] A. Champandard, 'Understanding behavior trees', *AIGameDev.com*, (2007). `http://aigamedev.com/open/article/bt-overview/` Last checked 2014-01-05.

[5] A. Champandard, 'Using decorators to improve behaviors', *AIGameDev.com*, (2007). `http://aigamedev.com/open/article/decorator/` Last checked 2014-08-10.

[6] D. Fu and R. Houlette-Stottler, 'The ultimate guide to FSMs in games', in *AI Game Programming Wisdom II*, 283–302, Charles River Media, (2004).