Symbolic Representation of Game World State: Toward Real-Time Planning in Games

Jeff Orkin
Monolith Productions, Inc.
Kirkland, WA 98033
http://www.lith.com
jorkin@blarg.net

Abstract

As the game development effort scales, AI developers are facing new challenges in terms of implementation, workflow, and game design. The needs of today's games are outgrowing the typical techniques of modeling behavior with Finite State Machines and Rule Based Systems. This paper argues that a regressive real-time planning system is better suited to address the challenges game developers are facing, and presents symbolic representation strategies that we have employed to allow planning in practice in today's games.

Introduction

With each generation of games, the bar is raised for AI, and game AI developers encounter new challenges. Faced with difficult problems, developers turn to academic formalisms for proven solutions. When Non-Player Characters (NPCs) needed to navigate game worlds, developers turned to A* [Stout96]. With the need to model complex behaviors, developers employed Finite State Machines (FSM) and Rule Based Systems (RBS).

Today game AI developers are encountering new challenges in terms of implementation, workflow, and game design. The scale of development is rapidly increasing, and there is a trend toward more open-ended gameplay. As production timelines, team sizes, and scope of game designs grow, AI developers need to share behaviors among NPCs or even between projects, and delegate AI responsibilities to other team members. Non-linear game designs require smarter NPCs that can learn and use complex reasoning to find alternate solutions to problems.

Demands of the next generation of games are starting to outgrow the common techniques of implementing NPC behaviors with explicit FSM transitions or RBS rules. The formalism of a real-time, regressive Goal Oriented Action Planning (GOAP) system addresses the issues that game AI developers are facing. Planning systems are not new to the field of AI, but have seldom been used, if at all, to model NPC behavior in commercial games. A GOAP system imposes a modular architecture that facilitates sharing behaviors among NPCs and projects. Atomic goals and actions of a GOAP system are easy to read and maintain, and can be sequenced and layered to create complex behaviors. The GOAP architecture lends itself to a separation of implementation and data that is ideal for the workflow of game developers. Regressively searching for plans in real-time affords NPCs opportunities to learn and find multiple solutions to problems.

If NPCs are expected to formulate their own plans in real-time, they need to be able to understand how actions and goals relate to one another. Representing the goal state and action preconditions and effects symbolically gives NPCs the means to understand the relationships between goals and actions. However, diluting increasingly realistic game worlds down to a set of symbols can be quite challenging. We first detail the benefits of a GOAP system, and then present strategies that we have employed to use planning in practice.

The Modular GOAP Architecture

A regressive GOAP system imposes an intuitive, modular architecture that maps well to the terminology game designers use to characterize NPC behavior. We modeled our architecture after the structure defined by the Planning Domain Definition Language [PDDL]. At the top level, an NPC has a set of goals that he or she wants to satisfy. The NPC tries to satisfy the goal or goals that are most relevant to his or her current situation, based on some prioritization. A planner searches for the sequence of actions that will satisfy the goal. Each action may have preconditions, which also need to be satisfied by the planner [Nilsson98].

Actions, too, may be prioritized to resolve cases where multiple actions have the same effects. This produces a layered system, where higher priority actions override others when appropriate. For example, the `AttackFromVehicle` action overrides the basic `Attack` action when an NPC is riding a vehicle. Both of these variations of the `Attack` action can satisfy an `EliminateEnemy` goal.

There is no explicit mapping between goals and actions, or actions to other actions. The planner searches for valid sequences of actions, and considers an action to be a valid neighbor if it has an effect that solves some unsatisfied symbol in the goal state, as described in [Orkin03]. This decoupling of goals and actions facilitates sharing of behaviors, as it allows developers to pick and choose which goals and actions will be available to an NPC, or to all NPCs in a game. Conversely, decoupling allows developers to choose goals or actions to exclude from the set of possible behaviors. This eliminates the proliferation of flags typically found in game AI code, for conditionals such as `CanSwim`, `CanFly`, `KicksDoors`, or `DestroysDoors`.

Sharing of behaviors is further facilitated by the simplicity of each individual goal and action. In order for the planner to algorithmically search for a satisfying plan, each goal and action needs to share a basic structure. The similarity between modules reduces the effects of coding style, making it easier for multiple developers to collaboratively implement complex behaviors.

Workflow with Designers

While it is possible to create data-driven FSMs and RBSs [Yiskis03, Champandard04], where the transitions or rules are separated from the implementation of the behaviors, this separation of code and data does not correlate well with the typical workflow of a game development team. Game Designers are responsible for designing spaces, placing scenarios, and directing the action from a high level. They are not concerned with micro-managing the decisions of NPCs, and not accustomed to thinking in terms of the logical expressions that drive state transitions or rules, let alone hierarchies of states or rules. Engineers are still responsible for the transitions or rules, even if they are separated into data.

A GOAP system, with decoupled actions and goals with no explicit connections, gives a separation between implementation and data that is better suited for the workflow of game developers. Engineers implement the atomic actions and goals, and embed the preconditions and effects within. Designers use data files to specify which goals and actions are available to different types of NPCs. This lets designers think about *what* the NPCs can do, without having to worry about the logic of *when* or *how* an NPC decides to do it. As long as Engineers specify appropriate preconditions and effects, NPCs will make use of the various goals and actions when it makes sense.

Our GOAP system bears similarity to, and was partially inspired by, the ABL reactive-planning language developed for Façade [Mateas02]. The separation between implementation and data is a key difference between our work and ABL. We took a toolkit approach, where engineers implement decoupled behaviors derived from primitive building blocks for goals and actions, and designers assign the behaviors to NPCs through data files. ABL is a language intended for use by designers to implement behaviors themselves.

Open-Ended Game-Play

If we free designers of the responsibility of micro-managing NPC behavior, they can concentrate on building more detailed worlds that offer more opportunities to both the player and the NPCs. The current trend towards open-ended worlds with less linear game-play requires NPCs with more intelligence and depth to their behavior. NPCs need a much wider range of behaviors than the simplistic patrol and pursue model.

A GOAP system gives NPCs the means to achieve the desired depth of behavior. This point is best illustrated with scenarios we have observed while developing our games. While testing a new feature that allows the player to steal an NPC's weapon, a developer was surprised when the NPC responded by running to grab a pipe off the wall and returning to flog the player with it! On another occasion, a developer who was being chased by an NPC ran into a room, closed the door, and blocked it with his body. After the pursuing NPC had no success kicking open the door, he out-smarted the player by diving through a nearby window and coming at the player from another direction.

These scenarios are not remarkable on their own, as the same results could be achieved with an FSM or RBS. The beauty of the GOAP solution is that it does not require any explicit rules to define how to handle blocked doors or stolen weapons. Reasonable solutions fall out for free based on the preconditions and effects of actions sequenced to satisfy goals. An empty-handed NPC with the goal of attacking an enemy satisfies the precondition of being armed by obtaining a weapon in any way possible. When an NPC finds his desired path obstructed by an impassable door, he abandons his plan and formulates a new one.

The preceding examples describe NPCs reasoning by selecting actions from a predetermined set, but it is also possible for NPCs to learn new actions as suggested in [Isla02]. NPCs might augment their set of possible actions after observing someone else performing an action. Additional effects of existing actions can be learned through observation or experimentation.

Symbolic Representation of Game World State

Despite the advantages of a GOAP system, the question remains, is it practical to encode an NPC's knowledge of the world symbolically?

We have employed a number of strategies to overcome the difficulties of symbolically representing the state of a detailed game world. These strategies have been applied to two First Person Shooters (FPS) in development at Monolith Productions, and have allowed us to ensure that our GOAP system supports real-time combat with up to ten NPCs at once. Meeting our performance requirements necessitates a system that can formulate plans quickly, and minimizes the frequency of new plan formulation. The ten NPC limit is imposed more by the graphics engine than by the AI systems.

We represent the state of the world with a data structure that consists of a fixed-size array of symbols, implemented as key-value pairs. Keys are represented by enumerated world properties. Values are a union of possible data types. Each NPC maintains its own symbolic view of the world through a world state member variable. The world state includes symbols for properties such as the NPC's position, weapon, amount of ammo, target object, and health of target object.

Designers assign a set of goals per type of NPC. Engineers implement the goals themselves. Each goal specifies the satisfaction state of some subset of the world state symbols. For example, and `EliminateEnemy` goal is satisfied when the NPC's current target has a health of zero.

Similar to the goals, actions are also implemented by engineers, and designers assign sets of actions per type of NPC. The effects of an action are specified as a subset of the world state symbols. The planner searches for actions that have effects that satisfy some goal. Actions may in turn have preconditions that need to be satisfied by the planner, also specified as a subset of the world state symbols.

In order to avoid combinatorial explosion while the planner searches for a valid sequence of actions to satisfy a goal, we hash our actions by the symbols they affect, and apply heuristics to guide the search. Hashing the actions by their effects allows the planner to quickly find candidate actions that may solve one of the unsatisfied symbols of the goal world state. The regressive search is implemented as an A* search that attempts to minimize the number of actions needed to solve the remaining unsatisfied goal world state symbols. [Orkin03] illustrates this process with a diagram.

While hashing and heuristics optimize the planner's search, we still do not want to plan more often than we need to. We only formulate a new plan when the current plan has been invalidated, or the most relevant goal has changed. The frequency of re-planning varies depending on the NPC's surroundings, but is far less frequent than every frame. The time between planner searches can sometimes be measured in minutes!

Let's examine our previous example of the NPC who grabs a pipe off the wall to attack the player. The NPC's most relevant goal is `EliminateEnemy`, which can be satisfied with an `Attack` action. The Attack action alone cannot satisfy the goal, because it has a precondition that the symbol `Armed` is set to `true`, and the NPC is currently bare-handed. The planner finds two candidate actions that can arm an NPC, `DrawWeapon` and `PickupWeapon`. `DrawWeapon` is not a valid choice, because the NPC does not have a holstered weapon to draw. `PickupWeapon` can be satisfied by grabbing the pipe off the wall, but it has an additional precondition that the NPC is standing at the world position of the pipe. The planner continues searching until it formulates the following plan to satisfy the `EliminateEnemy` goal:

```
Goto(pipe)
PickupWeapon(pipe)
Goto(target)
Attack(target)
```

Discussion of GOAP in this paper is limited to our experience with games of the FPS genre. It is easy to extrapolate how these same techniques could be applied to games of other genres, for example Real Time Strategy (RTS) games. Rather than planning the actions of individual units, a computer player could use a GOAP system to plan at a higher level, for example to select a sequence of actions for base building.

Mixing Symbolic and Non-Symbolic Preconditions

An NPC who formulates plans to eliminate threats and stay out of danger needs to know who is alive, who is dead, and who is aiming at whom. It would be prohibitively expensive in terms of memory and processing for each NPC to keep track of the state of everyone else. By taking an agent-centric point-of-view, the NPC can dilute information down to a minimal set of symbols. Rather than maintaining symbols for everyone's health and current target, the NPC can simply store a single symbol representing whether his current threat is alive, and another indicating whether the threat is currently aiming at the NPC. Outside of the planner, the NPC's sensors run custom processes to select the current threat, and monitor the threat's state.

The agent-centric strategy solves some problems, but there are still some action preconditions that cannot be precomputed by sensors. An NPC who wants to look at a disturbance needs to know if the point in space of the disturbance's origin is visible. An NPC who wants to dive into cover needs to know if there is enough room in front of him to play a dramatic animation. There are an infinite number of points in space that the NPC may be interested in, and a large number of animations that the NPC could potentially play. It would be impractical for a sensor to keep track of the visibility to every potentially interesting point in space, or the clearance available for the total translation of every animation.

The purpose of representing game state symbolically is to allow the planner to make connections between goals and actions, and actions to other actions. If there are preconditions that the planner is not intended to solve, they do not need to be represented by symbols. For instance, if the planner finds that some point in space is not visible to the NPC, there exists no action that will make it visible. A strategy for handling tests that need to be performed in real-time, like visibility tests of physics collision tests, is to allow actions to contain custom preconditions with arbitrary implementations, known as Context Preconditions. These preconditions may run any piece of code to check an action's validity in the context of the game world. Context Preconditions provide an alternative to symbolic representation of preconditions, and can be used to prune the search tree while planning. Similarly, actions may have Context Effects, which have arbitrary effects on the game world that do not concern the planner.

Future Work

There is a lot of room for improvement in the implementation of the planner. We imposed a number of limitations to get acceptable real-time performance from a GOAP system in practice. These limitations include only allowing conjunctions in action preconditions, and maintaining a static set of symbols in the planner's working memory. As GOAP systems become more prevalent in commercial games, improved implementations may allow preconditions to employ the full range of logical expressions, and dynamic sets of symbols in working memory.

Currently games handle action planning and path finding separately. A GOAP architecture shares much in common with a pathfinder, and it may be beneficial to merge these two systems. Aside from the implementation benefits of code sharing, NPCs could behave much more intelligently if they could factor their goals into their pathfinding heuristics. [Champandard03] presents similar ideas in a routing technique called Pathematics.

While we have acknowledged the potential for learning with a GOAP system, we have not yet applied these ideas to any games. Once planning becomes more common in games, the ability to learn new plans would be the next step.

Acknowledgements

Thanks to Alexander Nareyek and to the members of the Goal Oriented Action Planning (GOAP) Workgroup of the Artificial Intelligence Interface Standards Committee [AIISC] of the International Game Developers Association [IGDA].

References

[AIISC] AIISC of the AI SIG of the IGDA, http://www.igda.org/ai/

[Champandard03] Champandard, A. J., "Pathematics: Routing for Autonomous Agents," http://www.base-sixteen.com/Navigation/, 2003

[Champandard04] Champandard, A. J., *AI Game Development,* New Riders Publishing, 2004

[IDGA] International Game Developers Association, http://www.igda.org

[Isla02] Isla, D. and Blumberg, B., "New Challenges for Character-Based AI in Games," *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium,* AAAI Press, 2002

[Mateas02] Mateas, M. and Stern, A., "A Behavior Language for Story-based Believable Agents," http://www-2.cs.cmu.edu/~michaelm/publications/AI-IE2002.pdf

[Nilsson98] Nilsson, N. J., "STRIPS Planning Systems," *Artificial Intelligence: A New Synthesis,* Morgan Kaufmann Publishers, Inc., 1998

[Orkin03] Orkin, J., "Applying Goal-Oriented Action Planning to Games," *AI Game Programming Wisdom 2,* Charles River Media, 2003

[PDDL] Planning Domain Definition Language, http://www.informatik.uni-freiburg.de/~hoffmann/ipc-4/pddl.html

[Stout96] Stout, W. B., "Smart Moves: Intelligent Path-finding," *Game Developer Magazine*, CMP Media LLC, 1996.

[Yiskis03] Yiskis, E., "Finite-State Machine Scripting Language for Designers," *AI Game Programming Wisdom 2,* Charles River Media, 2003