Proceedings of the First
Artificial Intelligence and Interactive
Digital Entertainment Conference

Agent Architecture Considerations for Real-Time Planning in Games

Jeff Orkin
Monolith Productions, Inc.
jorkin@blarg.net

Abstract

Planning in real-time offers several benefits over the more typical techniques of implementing Non-Player Character (NPC) behavior with scripts or finite state machines. NPCs that plan their actions dynamically are better equipped to handle unexpected situations. The modular nature of the goals and actions that make up the plan facilitates re-use, sharing, and maintenance of behavioral building blocks. These benefits, however, come at the cost of CPU cycles. In order to simultaneously plan for several NPCs in real-time, while continuing to share the processor with the physics, animation, and rendering systems, careful consideration must taken with the supporting architecture. The architecture must support distributed processing and caching of costly calculations. These considerations have impacts that stretch beyond the architecture of the planner, and affect the agent architecture as a whole. This paper describes lessons learned while implementing real-time planning for NPCs for F.E.A.R., a AAA first person shooter shipping for PC in 2005.

Introduction

Planning in real-time is an alternative to the more common techniques of modeling character behavior with scripts or finite state machines (FSMs). Rather than traversing a predefined graph of state transitions, a planning Non-Player Character (NPC) searches for a sequence of actions to satisfy some goal.

(Orkin 2004) details the three main benefits of planning for game developers. NPCs that plan in real-time are better equipped to handle unexpected situations. Goals and actions provide modular building blocks of behavior that are easier to share, re-use, and maintain. The planning architecture provides separation between the data and implementation that maps well to the workflow of game development teams. These benefits, however, come at the cost of CPU cycles.

In this paper, we discuss considerations that must be taken into account when designing the agent architecture for NPCs that plan in real-time. These considerations are lessons learned over the past two years developing a planning-based AI for F.E.A.R. (Monolith 2005a), a AAA first person shooter shipping for PC in 2005.

In order for multiple NPCs to plan in real-time while sharing the processor with the animation, rendering and physics systems, NPCs need to minimize the number of planner search iterations and keep precondition evaluations as light weight as possible. We accomplished this by distributing the processing of costly preconditions over many frames, and caching results for the planner to inspect on-demand. Distributed processing and caching have impacts that extend far beyond the planner, affecting the agent architecture as a whole. Planning in real-time requires careful consideration of every aspect of the agent architecture.

Gameplay Description

Core gameplay in F.E.A.R. involves combat with between four and eight human, robotic, or supernatural enemies at a time. The upper limit on the number of characters is bound by the renderer. Humans form squads to advance in cover, suppress and flank, and search in formation. Less-than-human enemies work alone, cloaking, sticking to walls, lunging from the shadows, and leaping into the ceiling to ambush again. All of the combat behavior is performed by NPCs planning actions to satisfy goals. Squads behaviors are not directly implemented with the planner, but squads delegate tasks to NPCs, which they accomplish by planning actions. Squad members may autonomously choose to satisfy a higher priority goal rather than responding to the squad's task. For instance, an NPC will choose to run from a grenade to save his own life, rather than holding position and laying suppression fire to cover an ally.

Agent Architecture

Our agent architecture resembles the MIT Media Lab's C4 (Burke et al. 2001). An agent is composed of a blackboard, working memory, a handful of subsystems, and some number of sensors. Sensors detect changes in the world, and deposit these perceptions in dynamic working memory. The planner uses these perceptions to guide its decision-making, and ultimately communicates instructions to subsystems through the blackboard. Subsystems include the targeting, navigation, animation, and weapons systems.

Sensors perceive external visible and audible stimuli, as well as internal stimuli such as pain and desires. Some sensors are event-driven while others poll. Event-driven sensors are useful for recognizing instantaneous events like sounds and damage. Polling works better for sensors that need to extract information from the world. For example, a sensor may generate a list of potential tactical positions.

All knowledge generated by sensors is stored in working memory in a common format called a `WorkingMemoryFact`.

The decision-making mechanism is the primary difference between our architecture and C4, as we have replaced C4's "Action Tupples" with a real-time planner. When the sensors detect significant changes in the state of the world, the agent re-evaluates the relevance of his goals. Only one goal may be active at a time. When the most relevant goal changes, the agent uses the planner to search for the sequence of actions that will satisfy the goal. The planner validates action preconditions with `WorkingMemoryFacts`. An action activates by setting values on member variables of the blackboard. Subsystems update at some constant rate, and change their behavior according to instructions placed on the blackboard. For example, the `GotoTarget` action sets a new destination on the blackboard. The following update, the navigation system responds by finding a path to the new destination.

Distributed Processing

While searching for a sequence of actions to satisfy a goal, the planner needs to validate each candidate action's preconditions. Some of these preconditions may be costly to compute, relying on ray intersection or pathfinding procedures. The planner needs to complete the entire search within one frame without interrupting the overall performance of the game, so it cannot afford to do costly computations on-demand. Instead, we use sensors to amortize the cost of these expensive computations over many frames, and cache results in working memory.

An NPC may have any number of sensors. Each sensor updates every frame if necessary, but many update less frequently or only in response to an event. Sensors perform ray intersection tests, pathfinding, and other expensive operations such as sorting or analyzing tactical positions. `SensorSeeEnemy` is an example of a sensor that remains dormant until some visual stimuli arrives, at which time the sensor performs a ray intersection test. `SensorNodeCombat` is a sensor that polls the world three times per second, searching for potential places to hide or fire from covered positions. This sensor collects a list of potentially valid nodes, and then sorts them based on their distance from the NPC. Validity is based on radii associated with the nodes that must contain the NPC's current target.

Initially, we only allowed NPCs to update one sensor per agent update. This kept the processing load as light as possible, but we discovered that this was too restrictive, resulting in noticeably delayed reactions. We found a better solution is to give the sensor's update routine a Boolean return value, and return true only if the sensor has performed a significant amount of work. It is up to the programmer to determine what fits this criterion. Each frame, the NPC iterates over the sensors that do not need to update every frame, and continues to allow sensors to process until one returns true. All sensors that have an update rate of 0.0 update every frame, so these sensors generally perform lightweight operations.

In addition to evenly distributing the processing of multiple tasks, we also use sensors to incrementally process a single large task over many frames. When an NPC discovers a threat along the path to his current tactical destination, he crouches in place and re-evaluates possible destinations. Each frame, the `PassTarget` sensor finds the path to a known tactical position, and determines if the path is clear from danger. This process is repeated every frame until a safe route to a tactical position can be found. Distributed processing of sensors allows us to add intelligence to our NPCs that we could not previously support, due to the prohibitive cost of computing multiple paths per frame.

Constantly processing sensors certainly leads to more total processing than a system relying on lazy evaluation, but the overall load is more consistent and controlled. Sensors provide the planner with a constant stream of up-to-date data, eliminating the need for the planner to burden the CPU with processing beyond what is required by the search for a valid plan.

Caching

Sensors cache perceptions in working memory, in the form of `WorkingMemoryFacts`. All types of knowledge are stored in this common format. A `WorkingMemoryFact` is a record containing a set of associated attributes. Different subsets of attributes are assigned depending on the type of knowledge the fact represents. We have ten possible types of knowledge, including `Character`, `Object`, `Disturbance`, `Task`, `PathInfo`, and `Desire` Facts.

A Fact record contains a total of 16 member attributes. The most commonly assigned attributes are the position, direction, stimulus type, object handle, and update time. Each attribute has an associated confidence value that ranges from 0.0 to 1.0. Below is a pseudo-code representation of a `WorkingMemoryFact`:

```
WorkingMemoryFact
{
   Attribute<Vector3D>     Position
   Attribute<Vector3D>     Direction
   Attribute<StimulusType> Stimulus
   Attribute<Handle>       Object
   Attribute<float>        Desire
   ...
   float              fUpdateTime
}
```

Where each Attribute looks like this:

```
Attribute<Type>
{
   Type  Value
   float fConfidence
}
```

Confidence Values

The meaning of the confidence value associated with each attribute varies widely, but is unified conceptually. Confidence may represent an NPC's current stimulation, proximity to some object, or degree of desire. When applied to the `Stimulus` attribute, confidence represents how confident the NPC is that he is sensing some stimulus. For example, the confidence of a `Character` Fact's `Stimulus` attribute indicates the current level of visual stimulation that the NPC is aware of for this character. The confidence associated with the `Position` attribute represents the NPC's confidence in this location as a destination. A sensor that searches for tactical positions uses the confidence value of the `Position` attribute to indicate how close the node is to the NPC. The sensor sorts the nodes by distance and normalizes the confidence values to fall between 0.0 and 1.0. The node with the highest positional confidence is the closest. The intensity of an NPC's `Desire` attribute is characterized by his confidence that he is feeling this desire. The confidence value of a `Desire` Fact's `Desire` attribute indicates the NPC's current urge to satisfy some desire.

The planner can take advantage of the consistent knowledge representation and associated confidence values while validating preconditions. Working memory provides generic query functions to search for a matching Fact, a count of matching Facts, or a Fact with the maximum positional or stimulus confidence. Using this consistent interface, the planner can query working memory for the nearest tactical position, or the most visible enemy.

Centralized Knowledge

Caching all knowledge in a consistent format does not directly improve the efficiency of the planner, but rather provides a means of a global optimization. Facts could be hashed into bins based on the type of knowledge, or sorted in some manner. At a minimum, the most recently or most frequently accessed Fact could be cached for immediate retrieval. We did not find linear searches through working memory to be a performance bottleneck, so we did not apply any optimizations. As the number of Facts scales, it may be worthwhile to pursue query optimization.

Centralizing knowledge in working memory or on the blackboard provides the NPC with a persistent context that is often lost in FSM or scripted systems, as the NPC transitions between states or scripts. For instance, if the NPC eliminates a threat, he can immediately query working memory to determine who to target next. If the NPC stops climbing a ladder to fire at someone below, the knowledge that he was in the process of climbing a ladder persists on the blackboard. If knowledge is instead stored in member variables of states or scripts, information is often lost when an NPC's behavior changes.

Garbage Collection

One notable issue that caching introduces is that of garbage collection. Over the course of the game, working memory fills up with Facts that may be no longer useful to the NPC. It is unclear who is responsible for cleaning out irrelevant facts. Some of our sensors take the C++ approach to garbage collection, where the creator is responsible for destroying the Facts that it deposited. For instance, `SensorNodeCombat` clears existing Facts about tactical positions from working memory before creating new ones. This scheme does not work as well for sensors like `SensorHearDisturbance`, which creates Facts for each disturbance detected, and waits for the planner to respond by sending the NPC to investigate, and then clear `Disturbance` Facts upon completion. We left it in the programmer's hands to clean up Facts in different ways on a case by case basis. A safer approach may be to assign an expiration time to all Facts, and collect garbage periodically. Subsystems could extend expiration times where necessary.

Lightweight Planning

Outsourcing costly operations to sensors relieves the planner of much of the processing burden while searching. The planner's search operation itself is the last obstacle in reliably planning fast enough for real-time. We have taken a number of steps to minimize the search space, and optimize precondition validation.

We minimize the search space by placing strict limitations on the representation of action preconditions and effects. Preconditions and effects are represented symbolically, as a fixed sized array of key-value pairs. Appendix B contains a complete listing of our enumerated symbols, which are paired with four byte values. The value may be an integer, float, bool, handle, enum, or a reference to another symbol. The planner uses the precondition symbols to minimize the search space to only those actions that have effects matching the preconditions existing in the plan generated so far. In other words, the search only takes potentially fruitful branches rather than testing every combination of actions. Actions are stored in a hash table sorted by the effect symbols, so the planner can instantly find a list of candidates to satisfy some precondition. One action may be referenced by multiple hash table bins if it has multiple effects.

In addition to the symbolic preconditions, we further prune the search tree with "context preconditions." Actions may optionally provide a context precondition validation function of arbitrary code to prevent the action from further consideration. This validation function is where the planner queries values cached in working memory, or on the blackboard. For example, an NPC reacting to a disturbance checks his working memory to determine which disturbances he is aware of. If the NPC has detected a dangerous disturbance like an incoming grenade, he will consider the `ReactToDanger` or `EscapeDanger` actions, rather than the `InspectDisturbance` or `LookAtDisturbance` actions. All four of these actions have the symbolic effect of setting the `DisturbanceExists` symbol to false. An action may also have a context effect function, which runs arbitrary code after the action completes execution.

The planner represents its view of the current state of the world using the same array structure as that used to represent symbolic preconditions and effects. This makes it trivial to validate any precondition. We index the array by the enumerated symbols, so we can instantly determine if the planner believes that `WeaponArmed` is true, or `AtNode` equals "`node66`" in the current world state.

Storing the symbols in a fixed sized array of key-value pairs does restrict preconditions and world state representation in several ways. First, we have no means of describing who a symbol refers to. Next, preconditions are limited to a conjunction of clauses. Each symbol may only be used in one clause of an action's complete precondition expression. Finally, the total number of unique symbols needs to be managed, to minimize memory consumption since each node of the search tree contains a copy of the state of the world determined so far.

In our initial prototype, we tried to support more arbitrary precondition expressions, but were unable to get the required performance. We thought it might be useful to be able to express preconditions like:

```
((Bob AtNode "node33") ^
   (Joe AtNode "node43")) V
(Bill AtNode "node99")
```

Allowing variable numbers of clauses with repeated symbols led to dynamic memory allocations and slower precondition validation. With the fixed sized array indexed by symbol, we get instant look-ups of any value. Evaluating an arbitrary expression requires either interpreting expressions at runtime, or walking compiled expression trees. We developed strategies to work within the limitations of a fixed array of symbols, and have not found that these limitations to cause problems. It is possible that the limitations may be more problematic for other genres of games.

Our primary strategy for dealing with our limitations is the adoption of an "agent centric" representation. All symbols describe properties relative to the agent himself. This relieves us from having to keep track of who is associated with each symbol. An NPC does not need to consider the health of every potential enemy inside the planner. All that matters is finding a plan that satisfies `TargetIsDead`. Subsystems are responsible for the details. The target selection subsystem is responsible for constantly identifying the current target out of the known threats cached in working memory.

We deal with the limited total number of symbols by keeping each symbol as general as possible. Many symbols have evolved over the course of two years of development as needed. For instance, `ReactedToDamage` evolved from a Boolean symbol into `ReactedToEvent`, which describes an enumerated value for the event that the NPC reacted to.

Planning with A*

The planner conducts the actual search by running the A* algorithm. Using A* allows us to leverage the wealth of published optimizations developed for navigational path planning (Higgins 2002). Plus, A* supports guiding the search with heuristics and cost metrics. Our heuristic aims to minimize the number of unsatisfied symbols in the goal state. We apply a cost to actions to force A* to consider more specific actions before more general ones. For instance, try to `AttackFromCover` before the general purpose `Attack`.

Our use of A* is easiest to describe with a brief example. An NPC who wants to satisfy the `KillEnemy` goal needs to formulate a plan that results in setting the `TargetIsDead` symbol to true. All other symbols in the goal state are initially unset, as they are irrelevant to satisfying `KillEnemy`.

A* calculates the heuristic distance to the goal state as 1.0, because we have one unsatisfied symbol, and the actual distance so far is 0.0. The planner finds two candidate actions that have the effect of setting `TargetIsDead` to true: `Attack` and `AttackFromCover`. Both of these actions have a precondition that `WeaponLoaded` is true, but this is already the case, so this precondition symbol is ignored. `AttackFromCover` has an additional precondition that `AtNodeType` equals `kNode_Cover`.

When the planner computes the distances from these candidate actions, both have an actual distance of 1.0, but `AttackFromCover` has a heuristic distance of 1.0 while `Attack`'s heuristic distance is 0.0. This is due to the extra precondition symbol on `AttackFromCover`. The planner validates the cheapest plan formulated so far, and finds that the single `Attack` action is a valid plan for satisfying the `KillEnemy` goal.

We would like NPCs look intelligent by preferring to take cover before firing. Associating a cost with each action makes this possible. By giving the generic `Attack` action a cost of 5.0, while the other actions remain at the default cost of 1.0, we can guide A* towards our preferred plan. When we factor in the cost per action, the one step `Attack` plan has an actual distance of 5.0, while the actual distance of the two step plan `GotoNode`, `AttackFromCover` is 2.0. The NPC will fire from cover if possible.

Planning and Dynamic Behavior

The complexity added by distributed processing, caching, and planning is only worthwhile if it results in noticeably more dynamic behavior. NPCs that plan in real-time can handle subtlety and dependencies, and the biggest benefit comes from the ability to re-plan.

Subtlety

NPCs who detect that they are within the blast radius of incoming grenades run away, or crouch and flinch. We were surprised to find NPCs outside of the blast radius turning their heads to watch the grenades land. We did not intentionally implement this behavior. NPCs were trying to satisfy the `EscapeDanger` goal, but found the actions `EscapeDanger` and `ReactToDanger` were invalid due to the distance from the NPC to the grenade. The NPC found the next best candidate action, `LookAtDisturbance`, to set the `DisturbanceExists` symbol to false. This action was intended for NPCs reacting to disturbance sounds, but worked well as a reaction to distant flying grenades. These nuances add depth to behavior, and fall out for free when NPCs formulate their own plans in real-time.

Dependencies

The planner chains actions with other actions to satisfy dependencies in the form of preconditions. Some of these dependencies may originate from objects in the game world. Invisible game objects placed by designers to specify tactical positions may optionally have dependencies on other objects. For instance, an NPC needs to flip the table over before taking cover behind it. The planner handles dependencies like this by chaining additional actions. The final plan will look like this:

```
GotoNode(TableNode)
UseObject(Table)
GotoNode(NodeCover78)
AttackFromCover()
```

There is no limit to the number of dependencies that can be chained. Perhaps an NPC will need to activate a generator to turn on the power before operating a crane to drop a cargo container that he can use for cover.

Re-Planning

Our final example of dynamic behavior puts all of the pieces together while illustrating the power of re-planning. An NPC's `SeeEnemy` sensor detects a threat in a nearby office, and adds a Character fact to working memory. The target selection system responds to the new fact by specifying a new target on the blackboard. The NPC re-evaluates his goals, and selects `KillEnemy` as the most relevant.

The NPC formulates a plan to `GotoTarget` and `Attack`, using a melee weapon. While closing in on the threat, the enemy slams the door to the office and blocks it with his body. The closed door invalidates the path to the target, in turn invalidating the `GotoTarget` action.

After re-evaluating his goals, the NPC determines that `TraverseLink` is now the most relevant. He needs to find a way to traverse the NavMeshLink containing the door. He formulates a plan with a single action, `TraverseBlockedDoor`. The NPC kicks the door, but it still does not open. `TraverseBlockedDoor` has a context effect function that records that the door is still blocked in working memory.

The plan has completed, so the NPC once again re-evaluates his goals. The `TraverseLink` goal is no longer relevant due to the cached WorkingMemoryFact indicating that the door in front of him is impassable. He once again tries to satisfy the `KillEnemy` goal with the plan `GotoTarget` and `Attack`. The path planner takes into account the working memory fact indicating that the door is impassable, and finds an alternate route. The NPC runs around to the side of the office, dives through the window, and attacks the enemy. Diving through the window actually requires one more round of re-planning, because the window is a NavMeshLink traversable with the `TraverseLink` goal.

Future Work

Our experience applying real-time planning to games has met or exceeded our goals. We have been able to produce more dynamic behavior than previously possible with our technology, while keeping the system modular, maintainable, and reusable. In fact, we already have a second game in development using the planning based AI systems; *Condemned* for Xbox2 (Monolith 2005b). There is plenty of room for improvement in future generations of the system, however. Our planner has no scheduling facility, so all ordering of actions has to be enforced through more restrictive preconditions. This can lead to less generic, reusable actions. We could benefit from the introduction of a scheduler, enforcing orderings such as

DrawWeapon then GotoTarget, instead of GotoTarget then DrawWeapon. We could also benefit from adding a hierarchy to support compound actions. There are situations where designers always want a specific sequence of actions to occur, and it would be much simpler to specify an unbreakable compound action than to enforce a sequence of actions by chaining preconditions and effects. Finally, we have only applied planning to the actions of individual NPCs. In the future, generating plans for squads of NPCs in real-time could produce more dynamic, robust coordinated behaviors.

Appendix A: Symbols

```
kSymbol_AnimPlayed
kSymbol_AtNode
kSymbol_AtNodeType
kSymbol_AtTargetPos
kSymbol_DisturbanceExists
kSymbol_Idling
kSymbol_PositionIsValid
kSymbol_RidingVehicle
kSymbol_ReactedToWorldStateEvent
kSymbol_TargetIsAimingAtMe
kSymbol_TargetIsDead
kSymbol_TargetIsFlushedOut
kSymbol_TargetIsSuppressed
kSymbol_TraversedLink
kSymbol_UsingObject
kSymbol_WeaponArmed
kSymbol_WeaponLoaded
```

Appendix B: Actions

```
Animate
Attack
AttackFromNode
AttackFromVehicle
AttackGrenade
AttackGrenadeFromCover
AttackLunge
AttackMelee
AttackReady
BlindFireFromCover
DismountVehicle
DodgeRoll
DodgeShuffle
DrawWeapon
EscapeDanger
FlushOutWithGrenade
Follow
GetOutOfTheWay
GotoNode
GotoNodeOfType
GotoTarget
GotoValidPosition
HolsterWeapon
Idle
InspectDisturbance
InstantDeath
LookAtDisturbance
MountVehicle
ReactToDanger
Recoil
Reload
SuppressionFire
SurveyArea
TraverseBlockedDoor
TraverseLink
UseSmartObjectNode
```

References

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

Higgins, D. 2002. How to achieve Lightning-Fast A*. *AI Game Programming Wisdom,* 133-145. Hingham, Mass.: Charles River Media.

Burke, R., Isla, D., Downie, M., Ivanov, Y., and Blumberg, B. 2001. CreatureSmarts: The Art and Architecture of a Virtual Brain. In Proceedings of the Game Developers Conference, 147-166. San Jose, Calif.: International Game Developers Association.

Mateas, M. and Stern, A. 2002. A Behavior Language for Story-based Believable Agents. *Working notes of Artificial Intelligence and Interactive Entertainment*. AAAI Spring Symposium Series. Menlo Park, Calif.: AAAI Press.

Monolith Productions, Inc. 2005a. *F.E.A.R.*. Los Angeles, Calif.: Vivendi Universal Games.

Monolith Productions, Inc. 2005b. *Condemned.*. San Francisco, Calif.: Sega of America.

Nilsson, N. J. 1998. STRIPS Planning Systems. *Artificial Intelligence: A New Synthesis,* 373-400. San Francisco, Calif.: Morgan Kaufmann Publishers, Inc.

Orkin, J. 2003. Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom 2*, 217-228. Hingham, Mass.: Charles River Media.

Orkin, J. 2004. Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In AAAI Challenges in Game AI Technical Report, 26-30. Menlo Park, Calif.: AAAI Press.

Planning Domain Definition Language. 2002. http://planning.cis.strath.ac.uk/competition/