Three States and a Plan: The A.I. of F.E.A.R.

Jeff Orkin
Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group
http://www.jorkin.com

If the audience of GDC was polled to list the most common A.I. techniques applied to games, undoubtedly the top two answers would be A* and Finite State Machines (FSMs). Nearly every game that exhibits any A.I. at all uses some form of an FSM to control character behavior, and A* to plan paths. *F.E.A.R*. uses these techniques too, but in unconventional ways. The FSM for characters in *F.E.A.R*. has only three states, and we use A* to plan sequences of actions as well as to plan paths. This paper focuses on applying planning in practice, using *F.E.A.R*. as a case study. The emphasis is demonstrating how the planning system improved the process of developing character behaviors for *F.E.A.R*.

We wanted *F.E.A.R*. to be an over-the-top action movie experience, with combat as intense as multiplayer against a team of experienced humans. A.I. take cover, blind fire, dive through windows, flush out the player with grenades, communicate with teammates, and more. So it seems counter-intuitive that our state machine would have only three states.

Figure 1: Over-the-top action in *F.E.A.R.*

Three States

The three states in our state machine are `Goto`, `Animate`, and `UseSmartObject`. `UseSmartObject` is an implementation detail of our systems at Monolith, and is really just a specialized data-driven version of the `Animate` state. Rather than explicitly telling it which animation to play, the animation is specified through a `SmartObject` in the game database. For the purposes of this paper, we can just consider `UseSmartObject` to be the same as `Animate`. So that means we are really talking about an FSM with only two states, `Goto` and `Animate`!

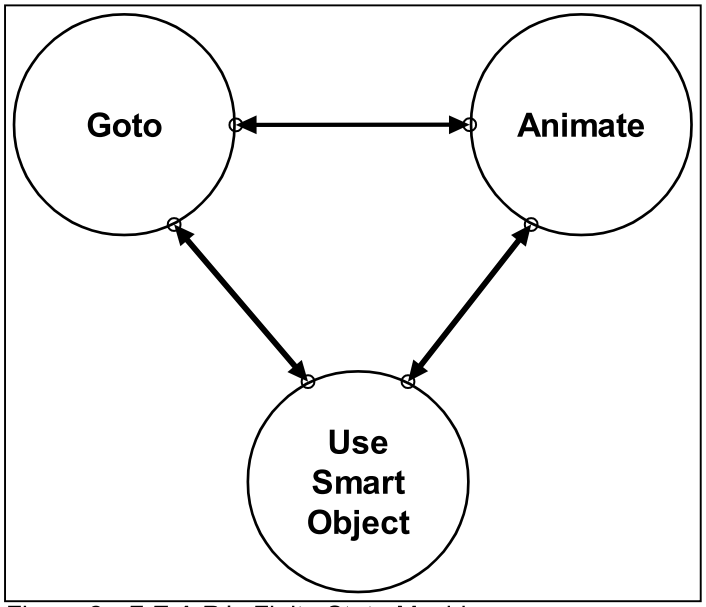


Figure 2: *F.E.A.R.*'s Finite State Machine

As much as we like to pat ourselves on the back, and talk about how smart our A.I. are, the reality is that all A.I. ever do is move around and play animations! Think about it. An A.I. going for cover is just moving to some position, and then playing a duck or lean animation. An A.I. attacking just loops a firing animation. Sure there are some implementation details; we assume the animation system has key frames which may have embedded messages that tell the audio system to play a footstep sound, or the weapon system to start and stop firing, but as far as the A.I.'s *decision-making* is concerned, he is just moving around or playing an animation.

In fact, moving is performed by playing an animation! And various animations (such as recoils, jumps, and falls) may move the character. So the only difference between `Goto` and `Animate` is that `Goto` is playing an animation while heading towards some *specific* destination, while `Animate` just plays the animation, which may have a side effect of moving the character to some arbitrary position.

The tricky part of modeling character behavior is determining when to switch between these two states, and what parameters to set. Which destination should the A.I. go to? Where is the destination? Which animation should the A.I. play? Should the animation play once, or should it loop? The logic determining when to transition from one state to another, and which parameters to specify need to live somewhere. This logic may be written directly into the C++ code of the states themselves, or may be external in some kind of table or script file, or it may be represented visually in some kind of graphical FSM visualization tool. However the logic is specified, in all of these cases it has to be specified manually by a programmer or designer.

Managing Complexity

For *F.E.A.R.*, this is where planning comes in. We decided to move that logic into a planning system, rather than embedding it in the FSM as games typically have in the past. As you will see in this paper, a planning system gives A.I. the knowledge they need to be able to make their own decisions about when to transition from one state to another. This relieves the programmer or designer of a burden that gets bigger with each generation of games.

In the early generations of shooters, such as *Shogo* (1998) players were happy if the A.I. noticed them at all and started attacking. By 2000, players wanted more, so we started seeing A.I. that can take cover, and even flip over furniture to create their own cover. In *No One Lives Forever* (*NOLF*) 1 and 2, A.I. find appropriate cover positions from enemy fire, and then pop randomly in and out like a shooting gallery. Today, players expect more realism, to complement the realism of the physics and lighting in the environments. In *F.E.A.R.*, A.I. use cover more tactically, coordinating with squad members to lay suppression fire while others advance. A.I. only leave cover when threatened, and blind fire if they have no better position.

With each layer of realism, the behavior gets more and more complex. The complexity required for today's AAA titles is getting unmanageable. Damian Isla's GDC 2005 paper about managing complexity in the Halo 2 systems gives further evidence that is a problem all developers are facing [Isla 2005]. This talk could be thought of as a variation on the theme of managing complexity. Introducing real-time planning was our attempt at solving the problem.

This is one of the main takeaways of this paper: It's not that any particular behavior in *F.E.A.R*. could not be implemented with existing techniques. Instead, it is the complexity of the combination and interaction of all of the behaviors that becomes unmanageable.

FSMs vs Planning

Let's compare FSMs to planning. An FSM tells an A.I. exactly how to behave in every situation. A planning system tells the A.I. what his goals and actions are, and lets the A.I. decide how to sequence actions to satisfy goals. FSMs are *procedural*, while planning is *declarative*. Later we will see how we can use these two types of systems together to complement one another.

The motivation to explore planning came from the fact that we had only one A.I. programmer, but there are lots of A.I. characters. The thought was that if we can delegate some of the workload to these A.I. guys, we'd be in good shape. If we want squad behavior in addition to individual unit behavior, it's going to take more man-hours to develop. If the A.I. are really so smart, and they can figure out some things on their own, then we'll be all set!

Before we continue, we should nail down exactly what we mean by the term *planning*. Planning is a formalized process of searching for sequence of actions to satisfy a goal. The planning process is called *plan formulation*. The planning system that we implemented for *F.E.A.R*. most closely resembles the STRIPS planning system from academia.

STRIPS Planning in a Nutshell

STRIPS was developed at Stanford University in 1970, and the name is simply an acronym for the STanford Research Institute Problem Solver. STRIPS consists of goals and actions, where goals describe some desired state of the world to we want to reach, and actions are defined in terms of preconditions and effects. An action may only execute if all of its preconditions are met, and each action changes the state of the world in some way. [Nilsson 1998, and Russell & Norvig 2002].

When we talk about *states* in the context of planning, we mean something different than the way we think about states in an FSM. In the context of an FSM, we're talking about procedural states, which update and run code to animate and make decisions; for example an `Attack` state or a `Search` state. In a planning system, we represent the state of the world as a conjunction of literals. Another way to phrase this is to say that we represent the state of the world as an assignment to some set of variables that collectively describe the world.

Let's say we wanted to describe a state of the world where someone is at home and wearing a tie. We can represent it as a logical expression, like this:

```
AtLocation(Home) ^ Wearing(Tie)
```

Or as an assignment to a vector of variable values, like this:

```
(AtLocation, Wearing) = (Home, Tie)
```

Looking at another example, if we are trying to represent the state of the world in the game *Lemonade Stand*, we'll need a vector of variables that keeps track of the weather, number of lemons, and amount of money we have. At different points in time, the weather may be sunny or rainy. We may have various numbers of lemons, and various amounts of money. It's worth noting that some of these variables have an enumerated discrete (e.g. sunny or rainy) set of possible values, while others are continuous (e.g. any number of lemons). Our goal state for this game will be one where we have lots of money. Any value assigned to weather and number of lemons will be Ok.

Now, let's look at an example of how the STRIPS planning process works. Let's say that Alma is hungry. Alma could call Domino's and order a pizza, but only if she has the phone number for Domino's. Pizza is not her only option, however; she could also bake a pie, but only she has a recipe. So, Alma's *goal* is to get to a state of the world where she is not hungry. She has two *actions* she can take to satisfy that goal: calling Domino's or baking a pie. If she is currently in a state of the world where she has the phone number for Domino's, then she can formulate the plan of calling Domino's to satisfy her hunger. Alternatively, if she is in the state of the world where she has a recipe, she can bake a pie. If Alma is in the fortunate situation where she has both a phone number and a recipe, either plan is valid to satisfy the goal. We'll talk later about ways to force the planner to prefer one plan over another. If Alma has neither a phone number nor a recipe, she is out of luck; there is no possible plan that can be formulated to satisfy her hunger. These examples show trivially simple plans with only one action, but in reality a plan may have an arbitrary number of actions, chained by preconditions and effects. For instance, if ordering pizza has a precondition that Alma has enough money, the satisfying plan may require first driving to the bank.

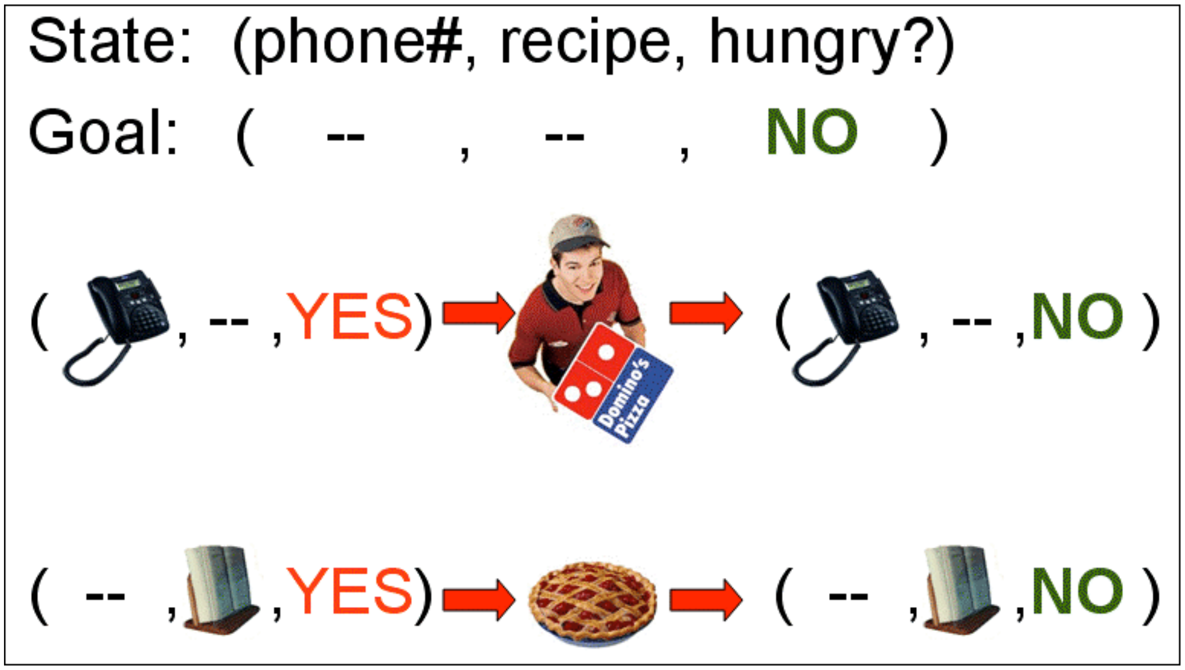


Figure 3: Two possible STRIPS plans to satisfy hunger.

We have previously described that a goal in STRIPS describes some desired state of the world that we want to reach. Now we will describe an action. A STRIPS action is defined by its preconditions and effects. The preconditions are described in terms of the state of the world, just like we defined our goal. The effects are described with list of modifications to the state of the world. First the Delete List removes knowledge about the world. Then, the Add List adds new knowledge. So, the effects of the `OrderPizza` action first remove knowledge that Alma is hungry, then adds knowledge that she is not hungry.

It may seem strange that we have to delete one assignment to the value of `Hungry`, and then add another, rather than simply changing the value. STRIPS needs to do this, because there is nothing in formal logic to constrain a variable to only one value. If the value of `Hungry` was previously `YES`, and the effect of `OrderPizza` is simply to add knowledge that `Hungry` is now set to `NO`, we will end up with a state of the world where `Hungry` is both `YES` and `NO`. We can imagine an action where this behavior is desirable. For example, the `Buy` action adds knowledge to the state of the world that we now own some object. We may be able to buy an arbitrary number of objects. Owning one object does not prevent us from owning another. The state of the world at some point may reflect that we own `coins`, a `key`, and a `sword`.

Back to our original example, there are two possible plans for feeding Alma. But what if instead we are planning for the cannibal Paxton Fettel? Neither of these plans that satisfied Alma's hunger will satisfy someone who only eats human flesh! We need a new `EatHuman` action to satisfy Fettel. Now we have three possible plans to satisfy hunger, but only two are suitable for Alma, and one is suitable for Paxton Fettel.

This is basically what we did for *F.E.A.R.*, but instead of planning ways to satisfy hunger, we were planning ways of eliminating threats. We can satisfy the goal of eliminating a threat by firing a gun at the threat, but the gun needs to be loaded, or we can use a melee attack, but we have to move close enough. So we've seen another way to implement behaviors that we could have already implemented with an FSM. *So what's the point?* It is easiest to understand the benefits of the planning solution by looking at a case study of how these techniques were applied to the workflow of modeling character behavior for *F.E.A.R*.

Case Study: Applying Planning to *F.E.A.R*.

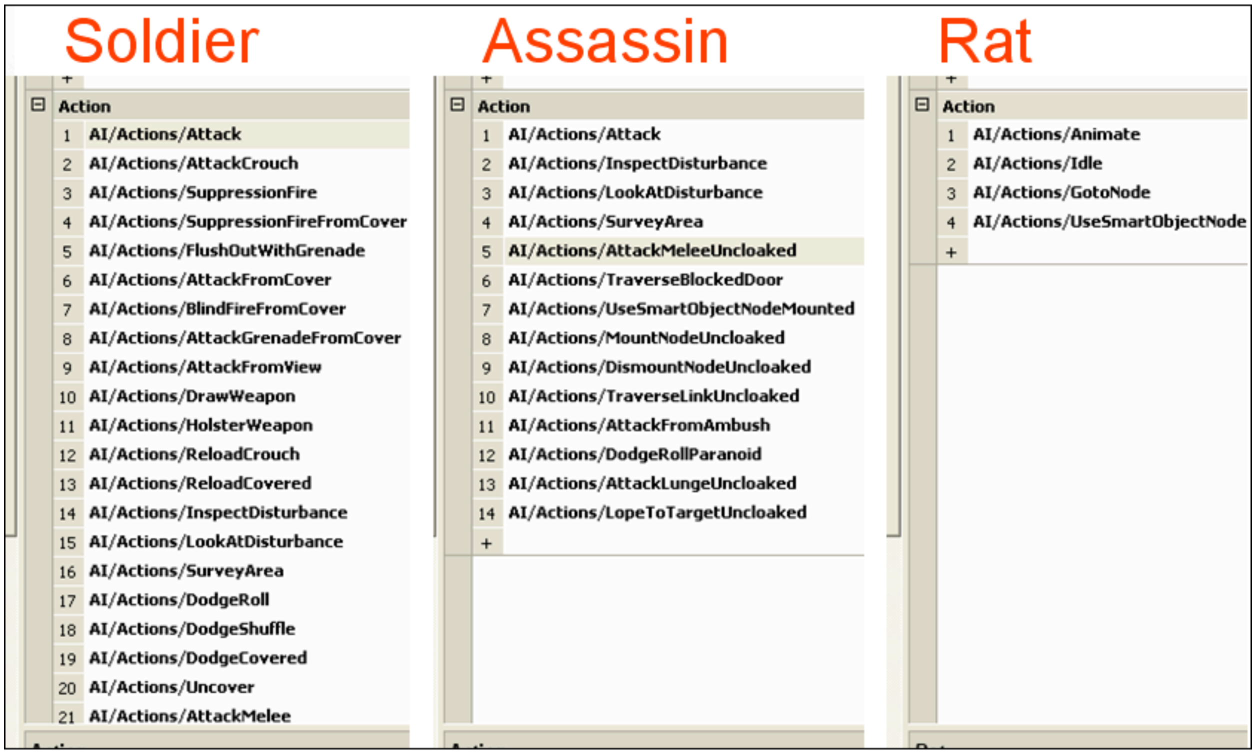
The design philosophy at Monolith is that the designer's job is to create interesting spaces for combat, packed with opportunities for the A.I. to exploit. For example, spaces filled with furniture for cover, glass windows to dive through, and multiple entries for flanking. Designers are not responsible for scripting the behavior of individuals, other than for story elements. This means that the A.I. need to autonomously use the environment to satisfy their goals.

If we simply drop an A.I. into the world in WorldEdit (our level editor), start up the game and let him see the player, the A.I. will do…. nothing. This is because we have not yet given him any goals. We need to assign a *Goal Set* to each A.I. in WorldEdit. These goals compete for activation, and the A.I. uses the planner to try to satisfy the highest priority goal.

We create Goal Sets in GDBEdit, our game database editor. For the purposes of illustration, imagine that we created a Goal Set named `GDC06` which contains only two goals, `Patrol` and `KillEnemy`. When we assign this Goal Set to our soldier in WorldEdit and run the game, he no longer ignores the player. Now he patrols through a warehouse until he sees the player, at which point he starts firing his weapon.

If we place an assassin in the exact same level, with the same `GDC06` Goal Set, we get markedly different behavior. The assassin satisfies the `Patrol` and `KillEnemy` goals in a very different manner from the soldier. The assassin runs cloaked through the warehouse, jumps up and sticks to the wall, and only comes down when he spots the player. He then jumps down from the wall, and lunges at player, swinging his fists.

Finally, if we place a rat in the same level with the `GDC06` Goal Set, we once again see different behavior. The rat patrols on the ground like the soldier, but never attempts to attack at all. What we are seeing is that these characters have the same goals, but different *Action Sets*, used to satisfy the goals. The soldier's Action Set includes actions for firing weapons, while the assassin's Action Set has lunges and melee attacks. The rat has no means of attacking at all, so he fails to formulate any valid plan to satisfy the `KillEnemy` goal, and he falls back to the lower priority `Patrol` goal.



Soldier

	Action
1	AI/Actions/Attack
2	AI/Actions/AttackCrouch
3	AI/Actions/SuppressionFire
4	AI/Actions/SuppressionFireFromCover
5	AI/Actions/FlushOutWithGrenade
6	AI/Actions/AttackFromCover
7	AI/Actions/BlindFireFromCover
8	AI/Actions/AttackGrenadeFromCover
9	AI/Actions/AttackFromView
10	AI/Actions/DrawWeapon
11	AI/Actions/HolsterWeapon
12	AI/Actions/ReloadCrouch
13	AI/Actions/ReloadCovered
14	AI/Actions/InspectDisturbance
15	AI/Actions/LookAtDisturbance
16	AI/Actions/SurveyArea
17	AI/Actions/DodgeRoll
18	AI/Actions/DodgeShuffle
19	AI/Actions/DodgeCovered
20	AI/Actions/Uncover
21	AI/Actions/AttackMelee

Assassin

	Action
1	AI/Actions/Attack
2	AI/Actions/InspectDisturbance
3	AI/Actions/LookAtDisturbance
4	AI/Actions/SurveyArea
5	AI/Actions/AttackMeleeUncloaked
6	AI/Actions/TraverseBlockedDoor
7	AI/Actions/UseSmartObjectNodeMounted
8	AI/Actions/MountNodeUncloaked
9	AI/Actions/DismountNodeUncloaked
10	AI/Actions/TraverseLinkUncloaked
11	AI/Actions/AttackFromAmbush
12	AI/Actions/DodgeRollParanoid
13	AI/Actions/AttackLungeUncloaked
14	AI/Actions/LopeToTargetUncloaked
+	

Rat

	Action
1	AI/Actions/Animate
2	AI/Actions/Idle
3	AI/Actions/GotoNode
4	AI/Actions/UseSmartObjectNode
+	

Figure 4: Three different Action Sets in GDBEdit.

Three Benefits of Planning

The previous case study illustrates the first of three benefits of a planning system. The first benefit is the ability to decouple goals and actions, to allow different types of characters to satisfy goals in different ways. The second benefit of a planning system is facilitation of layering simple behaviors to produce complex observable behavior. The third benefit is empowering characters with dynamic problem solving abilities.

Benefit #1: Decoupling Goals and Actions

In our previous generation of A.I. systems, we ran into the classic A.I. problem of "Mimes and Mutants." We developed our last generation A.I. systems for use in *No One Lives Forever 2* (*NOLF2*) and *Tron 2.0*. *NOLF2* has mimes, while *Tron 2.0* has mutants. Our A.I. system consisted of goals that competed for activation, just like they do in *F.E.A.R*. However, in the old system, each goal contained an embedded FSM. There was no way to separate the goal from the plan used to satisfy that goal. If we wanted any variation between the behavior of a mime and the behavior of a mutant, or between other character types, we had to add branches to the embedded state machines. Over the course of two years of development, these state machines become overly complex, bloated, unmanageable, and a risk to the stability of the project.

For example, we had out of shape policemen in *NOLF2* who needed to stop and catch their breath every few seconds while chasing. Even though only one type of character ever exhibited this behavior, this still required a branch in the state machine for the `Chase` goal to check if the character was out of breath. With a planning system, we can give each character their own Action Set, and in this case only the policemen would have the action for catching their breath. This unique behavior would not add any unneeded complexity to other characters.

The modular nature of goals and actions benefited us on *F.E.A.R*. when we decided to add a new enemy type late in development. We added flying drones with a minimal amount of effort by combining goals and actions from characters we already had. By combining the ghost's actions for aerial movement with the soldier's actions for firing weapons and using tactical positions, we were able to create a unique new enemy type in a minimal amount of time, without imposing any risk on existing characters.

There's another good reason for decoupling goals and actions as well. In our previous system, goals were self-contained black boxes, and did not share any information with each other. This can be problematic. Characters in *NOLF2* were surrounded by objects in the environment that they could interact with. For example someone could sit down at a desk and do some work. The problem was that only the `Work` goal knew that the A.I. was in a sitting posture, interacting wit the desk. When we shot the A.I., we wanted him to slump naturally over the desk. Instead, he would finish his work, stand up, push in his chair, and then fall to the floor. This was because there was no information sharing between goals, so each goal had to exit cleanly, and get the A.I. back into some default state where he could cleanly enter the next goal. Decoupling the goals and actions forces them to share information through some external working space. In a decoupled system, all goals and actions have access to information including whether the A.I. is sitting or standing, and interacting with a desk or some other object. We can take this knowledge into account when we formulate a plan to satisfy the `Death` goal, and slump over the desk as expected.

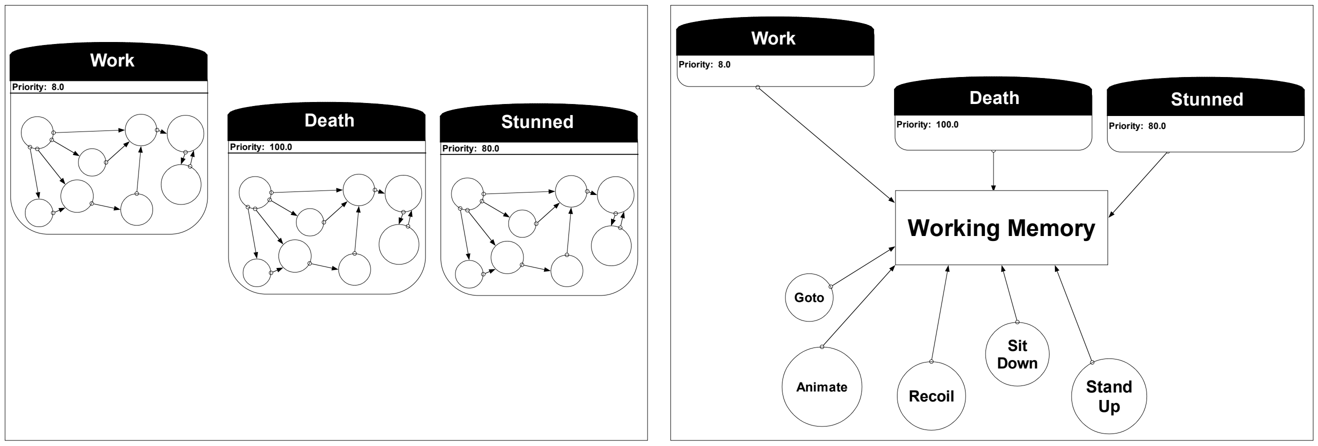


Figure 5: NOLF2 goals as black boxes (left) and F.E.A.R. decoupled goals and actions (right).

Benefit #2: Layering Behaviors

The second benefit of the planning approach is facilitating the layering of behaviors. You can think of the basic soldier combat behavior in *F.E.A.R.* as a seven layer dip. We get deep, complex tactical behavior by layering a variety of simple atomic behaviors. We wanted the *F.E.A.R.* A.I. to do everything for a reason. This is in contrast to the *NOLF2* A.I., which would run to valid cover and then pop in and out randomly like a shooting gallery. *F.E.A.R.* A.I. always try to stay covered, never leave cover unless threatened and other cover is available, and fire from cover to the best of their ability.

We start our seven layer dip with the basics; the beans. A.I. fire their weapons when they detect the player. They can accomplish this with the `KillEnemy` goal, which they satisfy with the `Attack` action.

We want A.I. to value their lives, so next we add the guacamole. The A.I. dodges when a gun is aimed at him. We add a goal and two actions. The `Dodge` goal can be satisfied with either `DodgeShuffle` or `DodgeRoll`.

Next we add the sour cream. When the player gets close enough, A.I. use melee attacks instead of wasting bullets. This behavior requires the addition of only one new `AttackMelee` action. We already have the `KillEnemy` goal, which `AttackMelee` satisfies.

If A.I. really value their lives, they won't just dodge, they'll take cover. This is the salsa! We add the `Cover` goal. A.I. get themselves to cover with the `GotoNode` action, at which point the `KillEnemy` goal takes priority again. A.I. use the `AttackFromCover` action to satisfy the `KillEnemy` goal, while they are positioned at a `Cover` node. We already handled dodging with the guacamole, but now we would like A.I. to dodge in a way that is context-sensitive to taking cover, so we add another action, `DodgeCovered`.

Dodging is not always enough, so we add the onions; blind firing. If the A.I. gets shot while in cover, he blind fires for a while to make himself harder to hit. This only requires adding one `BlindFireFromCover` action.

The cheese is where things really get exciting. We give the A.I. the ability to reposition when his current cover position is invalidated. This simply requires adding the `Ambush` goal. When an A.I.'s cover is compromised, he will try to hide at a node designated by designers as an `Ambush` node. The final layer is the olives, which really bring out the flavor. For this layer, we add dialogue that lets the player know what the A.I. is thinking, and allows the A.I. to communicate with squad members. We'll discuss this a little later.

The primary point we are trying to get across here is that with a planning system, we can just toss in goals and actions. We never have to manually specify the transitions between these behaviors. The A.I. figure out the dependencies themselves at run-time based on the goal state and the preconditions and effects of actions.

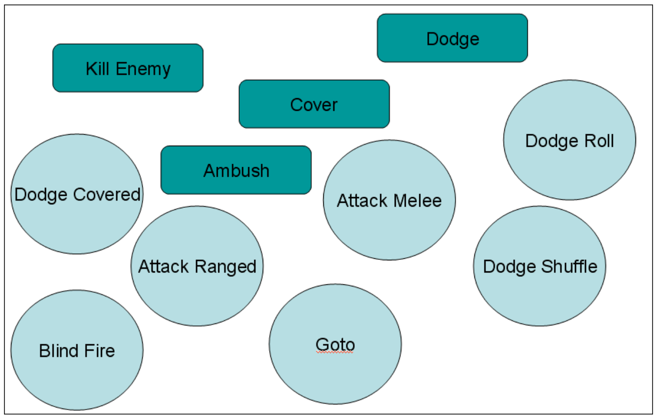
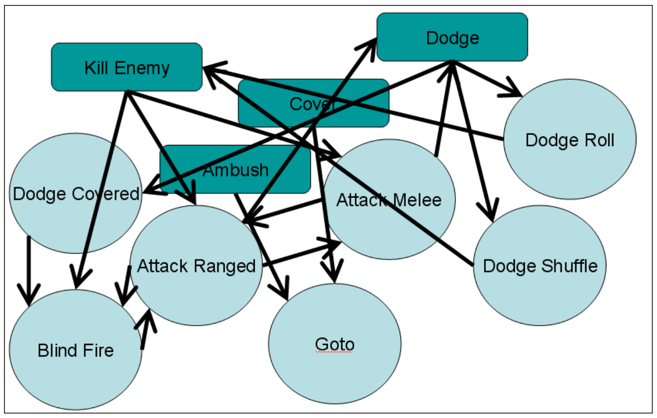


Figure 6: We do this.



But we never have to do this.

Late in development of *NOLF2*, we added the requirement that A.I. would turn on lights whenever entering a dark room. In our old system, this required us to revisit the state machine inside every goal and figure out how to insert this behavior. This was both a headache, and a risky thing to do so late in development. With the *F.E.A.R*. planning system, adding this behavior would have been much easier, as we could have just added a `TurnOnLights` action with a `LightsOn` effect, and added a `LightsOn` precondition to the `Goto` action. This would affect every goal that was satisfied by using the `Goto` action.

Benefit #3: Dynamic Problem Solving

The third benefit of a planning system is the dynamic problem solving ability that re-planning gives the A.I. Imagine a scenario where we have a patrolling A.I. who walks through a door, sees the player, and starts firing. If we run this scenario again, but this time the player physically holds the door shut with his body, we will see the A.I. try to open the door and fail. He then re-plans and decides to kick the door. When this fails, he re-plans again and decides to dive through the window and ends up close enough to use a melee attack!

This dynamic behavior arises out of re-planning while taking into account knowledge gained through previous failures. In our previous discussion of decoupling goals and actions, we saw how knowledge can be centralized in shared working memory. As the A.I. discovers obstacles that invalidate his plan, such as the blocked door, he can record this knowledge in working memory, and take it into consideration when re-planning to find alternative solutions to the `KillEnemy` goal.

Differences Between *F.E.A.R*. and STRIPS Planning

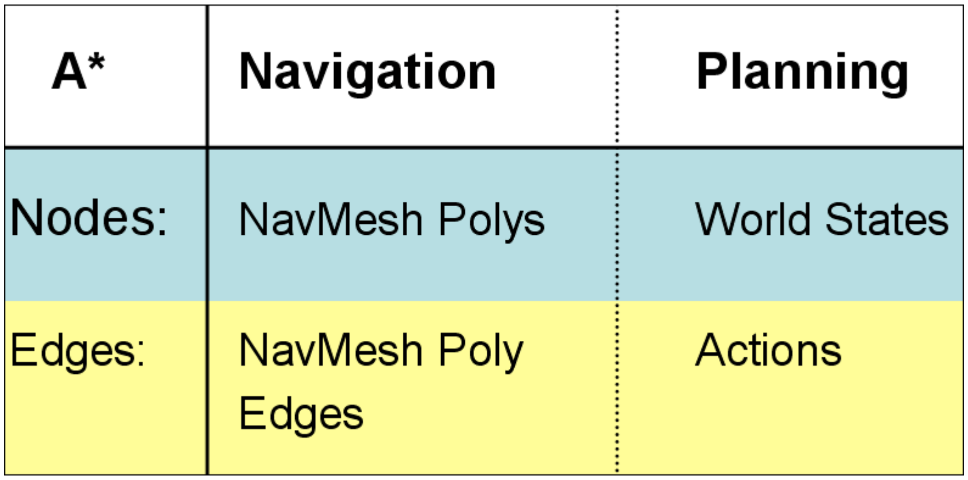
Now that we have seen the benefits of driving character behavior with a planning system, we will discuss how our system differs from STRIPS. We refer to our planning system as Goal-Oriented Action Planning (GOAP), as it was inspired by discussions in the GOAP working group of the A.I. Interface Standards Committee [AIISC]. We made several changes in order to make the planning system practical for a real-time game. These changes make the planner more efficient and controllable, while preserving the benefits previously described. Our system differs from STRIPS in four ways. We added a cost per action, eliminated Add and Delete

Lists for effects, and added procedural preconditions and effects [Orkin 2005, Orkin 2004, Orkin 2003]. The underlying agent architecture that supports planning was inspired by *C4* from the M.I.T. Media Lab's Synthetic Characters group, described in their 2001 GDC paper [Burke, Isla, Downie, Ivanov & Blumberg 2001], and is further detailed in [Orkin 2005].

Difference #1: Cost per Action

Earlier we said that if Alma has both the phone number and the recipe, either plan is valid to satisfy her hunger. If we assign a cost per action, we can force Alma to prefer one action over another. For example we assign the `OrderPizza` action a cost of 2.0, and the `BakePie` action a cost of 8.0. If she cannot satisfy the preconditions of ordering pizza, she can fall back to baking a pie.

This is where our old friend A* comes in! Now that we have a cost metric, we can use this cost to guide our A* search toward the lowest cost sequence of actions to satisfy some goal. Normally we think of A* as a means of finding a navigational path, and we use it in this way in *F.E.A.R.* too, to find paths through the navigational mesh. The fact is, however, that A* is really a general search algorithm. A* can be used to search for the shortest path through any graph of nodes connects by edges. In the case of navigation, it is intuitive to think of navigational mesh polygons as nodes, and the edges of the polygons as edges in the graph that connect that connect one node to another. In the case of planning, the nodes are states of the world, and we are searching to find a path to the goal state. The edges connecting different states of the world are the *actions* that lead the state of the world to change from one to another. So, we use A* for both navigation and planning in *F.E.A.R.*, and in each case we search through entirely different data structures. However, there are situations where we use both. For example, when an A.I. crawls under an obstacle, we first search for a navigational path, and then search for a plan that will allow the A.I. to overcome the obstacle on that path.



A*	Navigation	Planning
Nodes:	NavMesh Polys	World States
Edges:	NavMesh Poly Edges	Actions

Figure 7: Comparison of A* applied to navigation and planning.

Difference #2: No Add/Delete Lists

Our next modification to STRIPS is eliminating the Add and Delete Lists when specifying effects of actions. Instead of representing effects the way we described earlier with Add and

Delete Lists, we chose to represent both preconditions and effects with a fixed-sized array representing the world state. This makes it trivial to find the action that will have an effect that satisfies some goal or precondition. For example, the `Attack` action has the precondition that our weapon is loaded, and the `Reload` action has the effect that the weapon is loaded. It is easy to see that these actions can chain.

Our world state consists of an array of four-byte values. Here are a few examples of the type of variables we store:

```
TargetDead        [bool]
WeaponLoaded      [bool]
OnVehicleType     [enum]
AtNode            [HANDLE] -or- [variable*]
```

The two versions of the `AtNode` variable indicate that some variables may have a constant or variable value. A variable value is a pointer to the value in the parent goal or action's precondition world state array. For instance, the `Goto` action can satisfy the `Cover` goal, allowing the A.I. to arrive at the desired cover node. The `Cover` goal specifies which node to `Goto` in the array representing the goal world state.

The fixed sized array does limit us though. While an A.I. may have multiple weapons, and multiple targets, he can only reason about one of each during planning, because the world state array has only one slot for each. We use attention-selection sub-systems outside of the planner to deal with this. Targeting and weapon systems choose which weapon and enemy are currently in focus, and the planner only needs to concern itself with them.

Difference #3: Procedural Preconditions

It's not practical to represent everything we need to know about the entire game world in our fixed-sized array of variables, so we added the ability to run additional procedural precondition checks. For *F.E.A.R*. an action is a C++ class that has the preconditions both represented as an array of world state variables, and as a function that can do additional filtering. An A.I. trying to escape danger will run away if he can find a path to safety, or hunker down in place if he can't find anywhere to go. The run away action is preferable, but can only be used if the `CheckProceduralPreconditions`() function return true after searching for a safe path through the NavMesh. It would be impractical to *always* keep track of whether an escape path exists in our world state, because pathfinding is expensive. The procedural precondition function allows us to do checks like this on-demand only when necessary.

Difference #4: Procedural Effects

Similar to procedural preconditions, our final difference from STRIPS is the addition of procedural effects. We don't want to simply directly apply the effects that we've represented as world state variables, because that would indicate that changes are instantaneous. In reality it takes some amount of time to move to a destination, or eliminate a threat. This is where the planning system connects to the FSM. When we execute our plan, we sequentially activate our actions, which in turn set the current state, and any associated parameters.

The code inside of the `ActivateAction()` function sets the A.I. into some state, and sets some parameters. For example, the `Flee` action sets the A.I. into the `Goto` state, and sets some specific destination to run to. Our C++ class for an action looks something like this:

```
class Action
{
      // Symbolic preconditions and effects,
      // represented as arrays of variables.
      WORLD_STATE m_Preconditions;
      WORLD_STATE m_Effects;

      // Procedural preconditions and effects.
      bool CheckProceduralPreconditions();
      void ActivateAction();
};
```

Squad Behaviors

Now that we have A.I. that are smart enough to take care of the basics themselves, the A.I. programmer and designers can focus their energy on squad behaviors. We have a global coordinator in *F.E.A.R*. that periodically re-clusters A.I. into squads based on proximity. At any point in time, each of these squads may execute zero or one squad behaviors. Squad behaviors fall into two categories, simple and complex behaviors. Simple behaviors involve laying suppression fire, sending A.I. to different positions, or A.I. following each other. Complex behaviors handle things that require more detailed analysis of the situation, like flanking, coordinated strikes, retreats, and calling for and integrating reinforcements.

Simple Squad Behaviors

We have four simple behaviors. `Get-to-Cover` gets all squad members who are not currently in valid cover into valid cover, while one squad member lays suppression fire. `Advance-Cover` moves members of a squad to valid cover closer to a threat, while one squad member lays suppression fire. `Orderly-Advance` moves a squad to some position in a single file line, where each A.I. covers the one in front, and the last A.I. faces backwards to cover from behind. `Search` splits the squad into pairs who cover each other as they systematically search rooms in some area.

Simple squad behaviors follow four steps. First the squad behavior tries to find A.I. that can fill required slots. If it finds participants, the squad behavior activates and sends orders to squad members. A.I. have goals to respond to orders, and it is up to the A.I. to prioritize following those orders versus satisfying other goals. For example, fleeing from danger may trump following an order to advance. The behavior then monitors the progress of the A.I. each clock tick. Eventually either the A.I. fulfill the orders, or fail due to death or another interruption.

Let's look at the `Get-to-Cover` squad behavior as an example. Say we have a couple A.I. firing at the player from cover. If the player fires at one A.I. and invalidates his cover, the squad behavior can now activate. That is, it can find participants to fill the slots of one A.I. laying suppression fire, and one or more A.I. in need of valid cover, who have valid cover to go to. Note that the squad behavior does not need to analyze the map and determine where there is available cover. This is because each A.I. already has sensors keeping an up to date list of potentially valid cover positions nearby. All the squad behavior needs to do is select one node that the A.I. knows about, and ensure that other A.I. are not ordered to go to the same node. Once a node has been selected, the squad behavior sends orders to one A.I. to suppress, and orders the others to move to the valid cover positions. The A.I. re-evaluate their goals, and decide the highest priority goal is to begin following the orders. The squad behavior monitors their progress. If all A.I. fulfill their orders, and the repositioning A.I. has made it to a valid cover position, the squad behavior has succeeded. On the other hand, if the player throws a grenade, and invalidates the new cover position, the A.I. may re-evaluate his goals and decide it's a higher priority to flee than to follow orders. In this case, the A.I. flees and ends up somewhere unexpected, so the squad behavior fails.

Complex Squad Behaviors

Now let's look at our complex behaviors. The truth is, we actually did not have *any* complex squad behaviors at all in *F.E.A.R.* Dynamic situations emerge out of the interplay between the squad level decision making, and the individual A.I.'s decision making, and often create the illusion of more complex squad behavior than what actually exists!

Imagine we have a situation similar to what we saw earlier, where the player has invalidated one of the A.I.'s cover positions, and a squad behavior orders the A.I. to move to the valid cover position. If there is some obstacle in the level, like a solid wall, the A.I. may take a back route and resurface on the player's side. It appears that the A.I. is flanking, but in fact this is just a side effect of moving to the only available valid cover he is aware of.

In another scenario, maybe the A.I.s' cover positions are still valid, but there is cover available closer to the player, so the `Advance-Cover` squad behavior activates and each A.I. moves up to the next available valid cover that is nearer to the threat. If there are walls or obstacles between the A.I. and the player, their route to cover may lead them to come at the player from the sides. It appears as though they are executing some kind of coordinated pincher attack, when really they are just moving to nearer cover that happens to be on either side of the player. Retreats emerge in a similar manner.

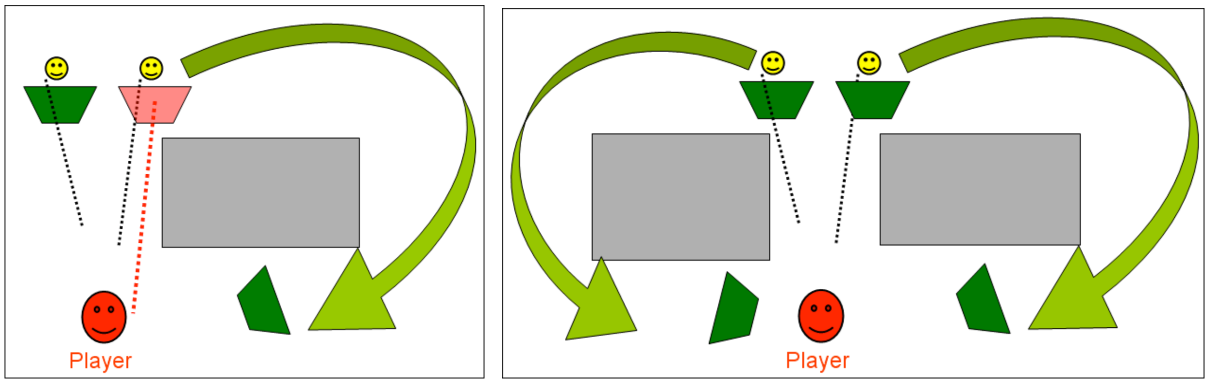


Figure 8: Complex squad behaviors emerging from the situation.

Squad Behavior Implementation

The design of our squad behavior system was inspired by Evans' and Barnet's 2002 GDC paper, "Social Activities: Implementing Wittgenstein" [Evans & Barnet 2002]. Our actual implementation of squad behaviors was not anything particularly formal. We only used the formalized STRIPS-like planning system for behavior of individual characters, and implemented something more ad-hoc for our squad behaviors. It is the *separation* between planning for individuals and planning for squads that is more important than any specific implementation.

A logical next step from what we did for *F.E.A.R*. would be to apply a formalized planning system to the squad behaviors. A developer interested in planning for squads may want to look at Hierarchical Task Network planning (HTN), which facilitates planning actions that occur in parallel better than STRIPS planning [Russell & Norvig 2002]. Planning for a squad of A.I. will require planning multiple actions that run in parallel. HTN planning techniques have been successfully applied to the coordination of Unreal Tournament bots [Muñoz-Avila & Hoang 2006, Hoang, Lee-Urban, & Muñoz-Avila 2005, Muñoz-Avila & Fisher 2004].

Squad Communication

There is no point in spending time and effort implementing squad behaviors if in the end the coordination of the A.I. is not apparent to the player. The squad behavior layer gives us an opportunity to look at the current situation from a bird's eye view, where we can see everyone at once, and find some corresponding dialogue sequence. Having A.I. speak to each other allows us to cue the player in to the fact that the coordination is *intentional*.

Vocalizing intentions can sometimes even be enough, without any actual implementation of the associated squad behavior. For example, in *F.E.A.R*. when an A.I. realizes that he is the last surviving member of a squad, he says some variation of "I need reinforcements." We did not really implement any mechanism for the A.I. to bring in reinforcements, but as the player progresses through the level, he is sure to see more enemy A.I. soon enough. The player's assumption is that the next A.I. encountered are the reinforcements called in by the previously speaking A.I., when in reality this is not the case.

Wherever possible, we try to make the vocalizations a dialogue between two or more characters, rather than an announcement by one character. For example, rather than having the A.I. cry out in pain when shot, we instead have someone else ask him his status, and have the injured A.I. reply that he's hit or alright. When the A.I. are searching for the player, rather than having one A.I. say "Where did he go?", we can have two A.I. in conversation where one asks the other if he sees anything. The other A.I. may respond with a negative, or call out a known or suspected position.

We also use dialogue to explain a lack of action. If an A.I. taking fire fails to reposition, he appears less intelligent. We can use dialogue to explain that he knows he needs to reposition, but is unaware of a better tactical position. The A.I. says "I've got nowhere to go!"

A gamer posting to an internet forum expressed that they he was impressed that the A.I. seem to actually understand each other's verbal communication. "Not only do they give each other orders, but they actually DO what they're told!" Of course the reality is that it's all smoke and mirrors, and really all decisions about what to say are made after the fact, once the squad behavior has decided what the A.I. are going to do.

Planning Beyond *F.E.A.R.*

It was rewarding to see that the A.I. was well received when *F.E.A.R.* shipped. Many people commented that the soldier A.I. reminded them of the marines in *Half Life* 1. *Half Life* shipped in 1998, and *F.E.A.R.* shipped in 2005. It seems that we haven't made much progress in seven years; and what's worse is that people seem happy about this! There has to be more we can do with game A.I. In the Cognitive Machines group at the M.I.T. Media Lab, we are looking at other applications of planning techniques that can impact future generations of games. The Cognitive Machines group uses robots and computer games as platforms for researching human-level language understanding. Our goal is to create robots and characters that can use language to communicate the way people do.

The *F.E.A.R.* combat dialogue system was completely separate from the action planning system. We manually hooked dialogue lines into the code in various places. It took a lot of trial and error to get A.I. saying the right things at the right times. For example, when an A.I.'s limbs have been completely severed by an explosion, there is really no reason for his ally to ask him "What's your status?" It's pretty obvious what his status is. Situations like this are not always obvious when you are looking at the C++ code trying to figure out where to insert dialogue lines. If we want an order of magnitude more dialogue, we to better inform the systems that choose what the A.I. say with the knowledge used to plan the rest of their behavior.

We are looking at how seamlessly integrating dialogue into the action planning system. If we think of dialogue lines as serving the same purpose as actions in a plan, then we should be able to formalize the lines in the same way we formalize actions. For anything the A.I. says, there are preconditions for why the A.I. should say it, and effects that the A.I. expects saying the line will have on the world. For example, if there is a grenade coming near an A.I.'s ally, the A.I. expects that shouting "Look out! Grenade!" will have the effect of his ally getting some distance form the grenade.

Complications arise when A.I. can accomplish the same effect by either acting or speaking. For example, a soldier may decide to open the door himself, or order a squad member to open the door. The decision of whether to speak or act is driven by cost calculations that need to take into consideration factors of the situation, such as who is closer to the door, and social factors, like who is the higher ranking member of the squad. A soldier does not usually shout orders to a superior.

Another aspect of planning is plan *recognition*. *F.E.A.R.* focused on A.I. cooperating with each other. If we want to make games more immersive, we would like A.I. to cooperate with the player; and in deeper ways than simply following the player around and shooting at things.

The problem is that the player's actions are unpredictable, so it is not always clear what the A.I. should do that would be useful to the player. If an A.I. can recognize the plan that the player is most likely trying to follow, the A.I. can then figure out how he can fit into that plan and be most useful to the player. [Gorniak & Roy 2005].

For example, if the player in an RPG is trying to open a locked door, he may be trying to follow a plan where he needs to take some gold to a forge, and has someone else keep the fire going while he forges a key. A hierarchical plan like this can also be represented as a context free grammar. We can then use parsing techniques originally developed for language processing to predict what plan the player is most likely trying to follow, and what action he might take next.

If we have seen the player pick up gold, he is likely to then go to the forge. If we have seen him get the gold, and go to the forge, we are even more confident that he is trying to create a key in order to unlock a door. Creating a key requires someone else to light a fire! So, we have found the most likely action that the player would like the A.I. take.

We have applied plan recognition to a simple scenario in an RPG prototype where there are three possible solutions to the puzzle of opening a locked door, and the plan recognition helps an A.I. understand ambiguous speech from the player, such as "Can you help me this?" The A.I. uses plan recognition to take his best guess at the intended meaning of the ambiguous language.

Conclusion

Real-time planning empowers A.I. characters with the ability to reason. Think of the difference between predefining state transitions and planning in real-time as analogous to the difference between pre-rendered and real-time rendered graphics. If we render in real-time, we can simulate the affects of lighting and perspective, and bring the gaming experience that much closer to reality. The same can be said of planning. By planning in real-time, we can simulate the affect of various factors on reasoning, and adapt behavior to correspond. With *F.E.A.R.*, we demonstrated that planning in real-time is a practical solution for the current generation of games. Moving forward, planning looks to be a promising solution for modeling group and social behaviors, increasing characters' command of language, and designing cooperative characters.

References

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

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

Evans, R., Barnet, T. 2002. Social Activities: Implementing Wittgenstein. *Proceedings of the Game Developers Conference*. International Game Developers Association.

Gorniak, P., Roy, D. 2005. Speaking with your Sidekick: Understanding Situated Speech in Computer Role Playing Games. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment.* AAAI Press.

Gorniak, P., Roy, D. 2005. Probabilistic Grounding of Situated Speech using Plan Recognition and Reference Resolution. *Seventh International Conference on Multimodal Interfaces*.

Hoang, H., Lee-Urban, S., Muñoz-Avila, H. 2005. Hierarchical Plan Representations for Encoding Strategic Game AI. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Isla, D. 2005. Handling Complexity in the Halo 2 AI. *Proceedings of the Game Developers Conference*. International Game Developers Association.

M.I.T. Media Lab, Cognitive Machines Group, http://www.media.mit.edu/cogmac/

Monolith Productions. 1998. *Shogo: Mobile Armor Division*. MC2-Microids.

Monolith Productions. 2000. *The Operative: No One Lives Forever*. Fox Interactive.

Monolith Productions. 2002. *No One Lives Forever 2: A Spy in H.A.R.M.'s Way*. Sierra Entertainment, Inc.

Monolith Productions. 2003. *Tron 2.0*. Buena Vista Interactive.

Monolith Productions. 2005. *F.E.A.R.: First Encounter Assault Recon.* Sierra Entertainment, Inc.

Muñoz-Avila, H., Fisher, T. Strategic Planning for Unreal Tournament Bots. 2004. *AAAI Challenges in Game AI Workshop Technical Report*. AAAI Press.

Muñoz-Avila, H. & Hoang, H. 2006. Coordinating Teams of Bots with Hierarchical Task Network Planning. *AI Game Programing Wisdom 3*. Charles River Media.

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

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

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

Orkin, J. 2005. Agent Architecture Considerations for Real-Time Planning in Games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Russell, S., Norvig, P. 2002. Artificial *Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Valve L.L.C. 1998. *Half-Life*. Sierra On-line, Inc.