# Applying Goal-Oriented Action Planning to Games

Jeff Orkin – Monolith Productions
*http://www.jorkin.com*

A number of games have implemented characters with goal directed decision-making capabilities. A goal-directed character displays some measure of intelligence by autonomously deciding to activate the behavior that will satisfy the most relevant goal at any instance. Goal-Oriented Action Planning (GOAP) is a decision-making architecture that takes the next step, and allows characters to decide not only what to do, but how to do it. But why would we want to empower our characters with so much freedom?

A character that formulates his own plan to satisfy his goals exhibits less repetitive, predictable behavior, and can adapt his actions to custom fit his current situation. In addition, the structured nature of a GOAP architecture facilitates authoring, maintaining, and re-using behaviors.

*No One Lives Forever 2: A Spy in H.A.R.M.'s Way (NOLF2)* [Monolith02] is one example of a game that contains goal-directed autonomous characters, but no planning capabilities. Characters in *NOLF2* constantly re-evaluate their goals, and select the most relevant goal to control their behavior. The active goal determines the character's behavior through a hard-coded sequence of state transitions.

This article explores how games can benefit from the addition of a real-time planning system, using problems encountered during the development of *NOLF2* to illustrate these points.

## Defining Terms

Before we can discuss the benefits of GOAP, we first need to define some terminology. An agent uses a *planner* to *formulate* a sequence of *actions* that will satisfy some *goal*. We need to define what we mean by the terms goal, action, plan, and formulate.

### *Goal*

A goal is any condition that an agent wants to satisfy. An agent may have any number of goals. Characters in *NOLF2* typically have about 25 goals. At any instant, one goal is active, controlling the character's behavior. A goal knows how to calculate its current relevance, and knows when it has been satisfied.

Goals in *NOLF2* fall into three categories: relaxed goals, investigative goals, and aggressive goals. Relaxed goals include passive goals such as `Sleep`, `Work`, and `Patrol`. Investigative goals include the more suspicious `Investigate` and `Search`. Aggressive goals are used for combat situations, like `Chase`, `Charge`, and `AttackFromCover`.

While conceptually similar, there is one key difference between the goals employed by *NOLF2* and the goals referred to by GOAP. *NOLF2*'s goals include an embedded plan. Once a goal is activated, the character runs through a pre-determined sequence of steps, hard-coded into the goal. The embedded plan can contain conditional branches, but these branches are predetermined at the time the goal is authored. Goals in GOAP do not include a plan. Instead, they simply define what conditions need to be met to satisfy the goal. The steps used to reach these satisfaction conditions are determined in real-time.

## *Plan*

The plan is simply the name for a sequence of actions. A plan that satisfies a goal refers to the valid sequence of actions that will take a character from some starting state to some state that satisfies the goal.

## *Action*

An action is a single, atomic step within a plan that makes a character do *something*. Some possible actions include `GotoPoint`, `ActivateObject`, `DrawWeapon`, `ReloadWeapon`, and `Attack`. The duration of an action may be short or infinitely long. The `ReloadWeapon` action will complete as soon as the character finishes a reload animation. The `Attack` action may continue infinitely, until the target is dead.

Each action knows when it is valid to run, and what it will do to the game world. In other words, an action knows its *preconditions* and *effects*. Preconditions and effects provide a mechanism for chaining actions into a valid sequence. For example, `Attack` has a precondition that the character's weapon is loaded. The effect of `ReloadWeapon` is that the weapon is loaded. It is east to see that `ReloadWeapon` followed by `Attack` is a valid sequence of actions. Each action may have any number of preconditions and effects.

A GOAP system does not replace the need for a finite-state machine (FSM) [Fu03], but greatly simplifies the required FSM. A plan is a sequence of actions, where each action represents a state transition. By separating the state transition logic from the states themselves, the underlying FSM can be much simpler. For instance, `Dodge` and `ReloadWeapon` are different actions that both set the character's state to `Animate`, and specify an animation to play. Rather than having a `Patrol` or `Wander` state, a GOAP system can formulate a plan that instructs the character to use to the `Goto` state to move between a number of patrol points. Ultimately, the `Goto` and `Animate` states cover most of things that characters do; they just do them for different reasons. Actions define when to transition into and out of a state, and what happens to the game world as a result of this transition.

## *Plan Formulation*

A character generates a plan in real-time by supplying some goal to satisfy to a system called a *planner*. The planner searches the space of actions for a sequence that will take the character from his starting state to his goal state. This process is referred to as *formulating* a plan. If the planner is successful, it returns a plan for the character to follow to direct his behavior. The character follows this plan to completion, invalidation, or until another goal becomes more relevant. If another goal activates, or the plan in-progress becomes invalid for any reason, the character aborts the current plan and formulates a new one.

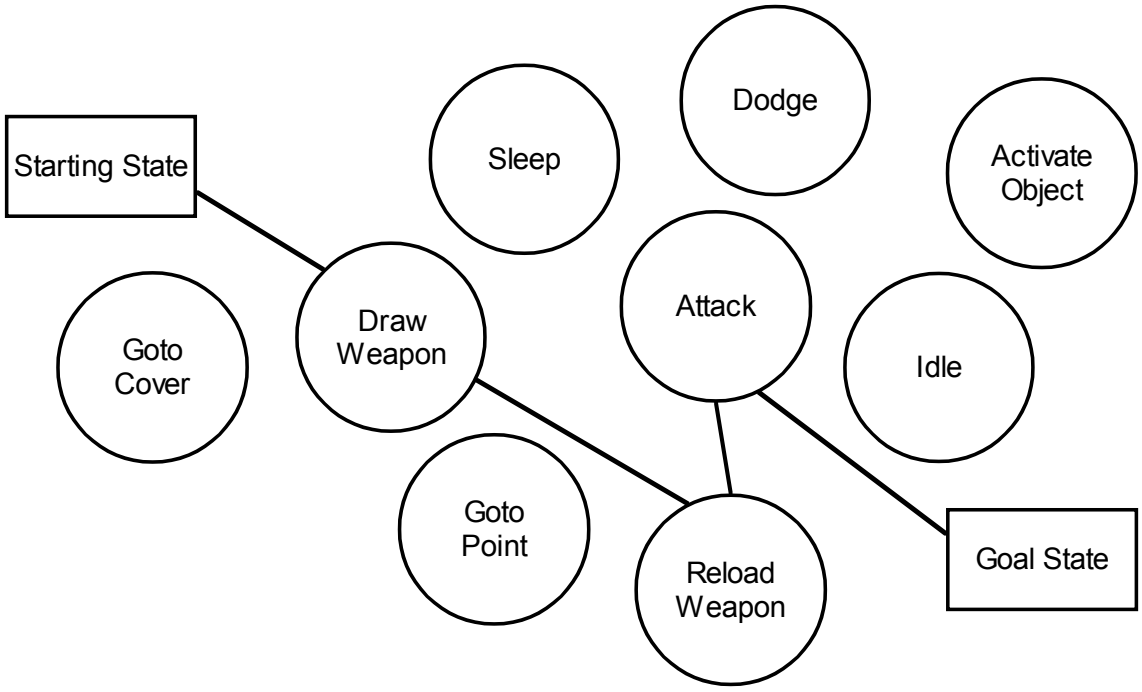


Figure 1. The plan formulation process.

Figure 1 depicts an abstract illustration of the planning process. The rectangles represent the start and goal states, and each circle represents an action. The goal in Figure 1 is to kill an enemy. Therefore, the goal state is the state of the world in which the enemy is dead. The planner needs to find a sequence of actions for the character that will take the world from a state in which is the enemy is alive to a state in which the enemy is dead.

This process looks suspiciously like pathfinding! In a sense that's exactly what it is. The planner needs to find a path through the space of actions that will take the character from his starting state to some goal state. Each action is a step on that path that changes the state of the world in some way. Preconditions of the actions determine when it is valid to move from one action to the next.

In many cases, more than one valid plan exists. The planner only needs to find one of them. Similar to navigational pathfinding, the planner's search algorithm can be provided with hints to guide the search. For example, costs can be associated with actions, leading the planner to find the least costly sequence of actions, rather than any arbitrary sequence.

All of this searching sounds like a lot of work. Is it worth it? Now that we have defined our terms, we can discuss the benefits of this decision-making process.

# The Benefits of GOAP

There are benefits at both development and runtime. Characters in the game can exhibit more varied, complex, and interesting behaviors using GOAP. The code behind the behaviors is more structured, re-usable, and maintainable.

### *Benefits to Runtime Behavior*

A character that determines his own plan at runtime can custom fit his actions to his current surroundings, and dynamically finds alternate solutions to problems. This is best illustrated with an example.

Imagine that character X detects an enemy who he wants to eliminate. Ordinarily, the best course of action would be to draw a loaded weapon and fire it at the enemy. In this case however, X has no weapon, or maybe no ammunition, so he needs to find an alternate solution. Luckily, there is a mounted laser nearby that X can use to blast the enemy. X can formulate a plan to go to the laser, activate it, and use it to zap someone. The plan might look like:

```
GotoPoint(laser)
ActivateObject(laser)
MountedAttack(laser)
```

Problem solved! Character X fed the planner his `KillEnemy` goal, and the planner formulated a valid plan to satisfy it. The planner hit a dead-end when it tried to include the action `DrawWeapon` in its sequence, because `DrawWeapon` has a precondition that the character has a weapon. Instead it found an alternate plan that does not require the character to have a weapon.

But wait! What if the mounted laser requires power, and the generator is turned off? The planner can handle this situation just as well. The valid plan for this situation involves first going to the generator and turning it on, and then using the laser. The plan might look like:

```
GotoPoint(generator)
ActivateObject(generator)
GotoPoint(laser)
ActivateObject(laser)
MountedAttack(laser)
```

The GOAP decision-making architecture allows character X to handle dependencies that may not have been anticipated at the time of the behavior development.

## *Benefits to Development*

Handling every possible situation in hand-authored code or scripts gets hairy fast. Imagine the code for a `KillEnemy` goal with an embedded plan that handles the situation described previously. The goal's embedded plan needs to handle characters with or without weapons, plus finding, navigating to, and activating alternate means of destruction.

It might be tempting to break down the `KillEnemy` goal into smaller goals, such as `KillEnemyArmed`, `KillEnemyMounted`. This is essentially what we did for *NOLF2*, but the proliferation of goals has its own problems. More goals means more code to maintain, and revisit with each design change.

Seemingly innocent design additions caused headaches during the development of *NOLF2*. Examples of these additions included drawing and holstering weapons, turning on lights when entering dark rooms, and activating security keypads before opening doors. Each one of these additions required revisiting the code of every goal to ensure that the embedded plan could handle the new requirements.

GOAP offers a much more elegant structure that better accommodates change. The addition of design requirements is handled by adding actions, and preconditions to related actions. This is more intuitive, and touches less code than revisiting every goal. For example, requiring characters to turn on lights in dark rooms before entering can be handled by adding a precondition to the `GotoPoint` action that requires the lights to be on in the destination. Due to this precondition, every goal that results in moving the character transparently handles turning on the lights in dark rooms before entering.

Furthermore, GOAP provides the guarantee of valid plans. Hand-coded embedded plans can contain mistakes. A developer might code a sequence of actions that cannot follow each other. For example, a character might be instructed to fire a weapon, without ever being told to first draw a weapon. This situation cannot arise in a plan dynamically generated through a GOAP system, because the preconditions on the actions prevent the planner from formulating an invalid plan.

## *Benefits to Variety*

The structure imposed by GOAP is ideal for creating a variety of character types who exhibit different behaviors, and even share behaviors across multiple projects. The planner is provided with a pool of actions from which to search for a plan. This pool does not necessarily have to be the complete pool of all existing actions. Different character types may use subsets of the complete pool, leading to a variety of behaviors.

*NOLF2* has a number of different character types including Soldiers, Mimes, Ninjas, Super Soldiers, and Bunnies. We tried to share as much of the AI code as possible between all of these character types. This sometimes led to undesirable branching in the code for behaviors. One instance of undesirable branching was related to how characters

handle closed doors. A human stops at a door, opens it, and walks through, while a cyborg Super Soldier smashes the door off its hinges and continues walking. The code that handles going through a door needs a branch to check if this is a character that smashes doors.

A GOAP system can handle this situation more elegantly, by providing each character type with a different action to accomplish the same effect. A human character can use the `OpenDoor` action, while a Super Soldier uses the `SmashDoor` action. Both actions have the same effect. They both open a path that was previously blocked by a door.

There are other solutions to the smashing versus opening the door problem, but none as flexible as the GOAP solution. For example, `OpenDoor` and `SmashDoor` could be states derived from a base `HandleDoor` class. Using a state class hierarchy like this, a designer can assign states to slots, such as the `HandleDoor` slot, creating character types that handle doors in different ways. But what if we want a character that opens doors when relaxed, and smashes doors when agitated? The state class hierarchy solution needs an external mechanism to swap out the state in `HandleDoor` slot under certain conditions. The GOAP solution allows a character to have both the `OpenDoor` and `SmashDoor` actions at all times. An additional precondition on each action for the required mood allows the character to select the appropriate action to handle the door in real-time, without any external intervention.

# Implementation Considerations

Now that you have seen the benefits, and are excited about the prospect of applying GOAP to games, you need to be aware of some good and bad news. The bad news is that there are a couple of challenges involved in implementing a GOAP system. The first challenge is determining the best method for searching the space of actions. The second challenge is one of world representation. In order to formulate a plan, the planner must be able to represent the state of the world in a compact and concise form. Both of these topics are large areas of research in academia, and a complete discussion of them is beyond the scope of this article. The good news is that we can stick to simple solutions for the domain of games. The rest of this article presents some reasonable solutions to these challenges that make sense for game development.

## *Planner Search*

Earlier we observed that the planning process is remarkably similar to navigational pathfinding. These processes are so similar, in fact, that we can use the same algorithm for both of them! The planner's search can be driven by an algorithm that most game AI developers are already intimately familiar with; namely A*. Though many game developers think of A* as a pathfinding algorithm, it is actually a general-purpose search algorithm. If A* is implemented in a modular fashion, like that described by [Higgins02a], the bulk of the code for the algorithm can be shared between the navigation

system and the planner. The planner just needs to implement its own classes for A*'s node, map, and goal.

The A* algorithm requires the calculation of the cost of a node, and the heuristic distance from a node to the goal. Nodes in the planner's search represent states of the world, with edges representing actions between them. The cost of a node can be calculated as the sum of the costs of the actions that take the world to the state represented by the node. The cost of each action may vary, where lower cost actions are more preferable. The heuristic distance can be calculated as the sum of the unsatisfied properties of the goal state.

We have two choices when searching with A*. We can search forward, starting at our current state and searching for a path to the goal state, or we can search backwards from the goal to the starting state. Let's first examine how a forward search would work for the previously described situation where an unarmed character wants to eliminate an enemy with a mounted laser that requires power. The forward search will first tell the character to go to the laser with the `GotoPoint` action, and then tell the character to use the laser with the `AttackMounted` action. The precondition on `AttackMounted` will fail if the power is off. It will take an exhaustive brute force search to come up with the valid plan that will first send the character to turn on the generator, and then use the laser.

A regressive search is more efficient and intuitive. Searching backwards will start at the goal, and find that the `AttackMounted` action will satisfy the goal. From there, the search will continue for actions that will satisfy the preconditions of the `AttackMounted` action. The preconditions will lead the character step-by-step to the final plan of first activating the generator, and then using the laser.

### *World Representation*

In order to search the space of actions, the planner needs to represent the state of the world in some way that lets it easily apply the preconditions and effects of actions, and recognize when it has reached the goal state. One compact way to represent the state of the world is with a list of world property structures that contain an enumerated attribute key, a value, and a handle to a subject.

```
struct SWorldProperty
{
   GAME_OBJECT_ID hSubjectID;
   WORLD_PROP_KEY eKey;

   union value
   {
      bool   bValue;
      float  fValue;
      int    nValue;
      ...
   };
```

```
};
```

So, if we wanted to describe the state of the world that will satisfy the `KillEnemy` goal, we would supply the goal state with a property that looks like this:

```
SWorldProperty Prop;
Prop.hSubjectID = hShooterID;
Prop.eKey = kTargetIsDead;
Prop.bValue = true;
```

Representing every aspect of the world in this way would be an overwhelming and impractical task, but this is unnecessary. We only need to represent the minimal set of properties of the world state that are relevant to the goal that the planner is trying to satisfy. If the planner is trying to satisfy the `KillEnemy` goal, it does not need to know the shooter's health, current location, or anything else. The planner does not even need to know whom the shooter is trying to kill! It just needs to find a sequence of actions that will lead to this shooter's target getting killed, whomever that target may be.

As the planner adds actions, the goal state grows as preconditions of the actions are appended to the goal's satisfaction state. Figure 2 illustrates a planner's regressive search to satisfy the `KillEnemy` goal. The search successfully completes when the current state matches the goal state. The goal state grows as actions add their preconditions. The current state grows accordingly as the planner searches for additional actions that will satisfy the additional goal properties.

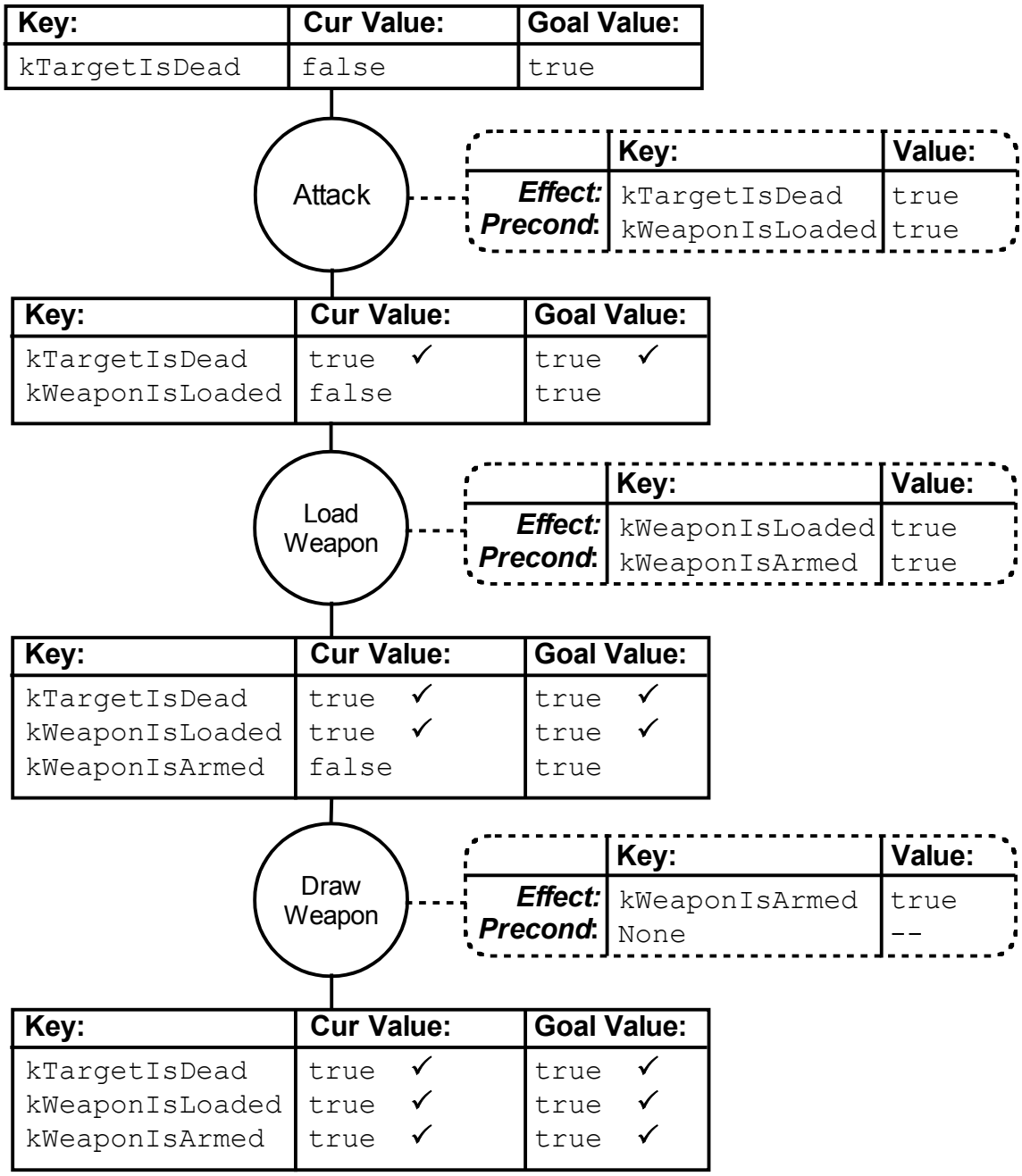

Figure 2. The planner's regressive search.

In each step of the regressive search, the planner tries to find an action that has an effect that will satisfy one of the unsatisfied goal conditions. A property of the world is considered unsatisfied when the goal state's property has a different value from the current state's property. Often, actions that solve one of the unsatisfied conditions add

additional preconditions to be satisfied. When the search completes, we can see that a valid plan to satisfy the `KillEnemy` goal is:

```
DrawWeapon
LoadWeapon
Attack
```

The planning example illustrated in Figure 2 consists of actions that have constant Boolean values for preconditions and effects, but it is important to point out that preconditions and effects can also represented by variables. The planner solves for these variables as it regresses from the goal. Variables add power and flexibility to the planner, as it can now satisfy more general preconditions. For instance, a `Goto` action with the effect of moving a character to a variable destination is far more powerful than a `Goto` action that moves to a constant, predetermined location.

Actions can use the previously described world state representation to represent their preconditions and effects. For example, the constructor for the Attack action defines its preconditions and effects like this:

```
CAIActionAttack::CAIActionAttack()
{
   m_nNumPreconditions = 1;
   m_Preconditions[0].eKey = kWeaponIsLoaded;
   m_Preconditions[0].bValue = true;

   m_nNumEffects = 1;
   m_Effects[0].eKey = kTargetIsDead;
   m_Effects[0].bValue = true;
}
```

Actions only need to specify preconditions in this symbolic manner if they are going to factor into the planner's search. There may be additional preconditions that can be called *context* preconditions. A context precondition is something that needs to be true, but that the planner will never try to satisfy. For example, the `Attack` action may require the target to be within some distance and field of view. This is a more complex check than can be represented with an enumerated value, and the planner has no actions that will make this true if it is not true already.

When the planner is searching for actions, it calls two separate functions. One function checks the symbolic planner preconditions, and the other checks freeform context preconditions. The context precondition validation function can contain any arbitrary piece of code that results in a Boolean value. Since the context preconditions for an action will be re-evaluated every time the planner attempts to add this action to a plan, it is important to minimize the processing required for this validation. Possible optimizations include caching results from previous validations, and looking up values that are periodically computed outside of the planner.

### *Planner Optimization*

Some consideration must be given to optimizing the planner's search. The complexity of formulating a plan will increase as the number of actions, and preconditions on actions grow. We can attack this problem with some of the same strategies used to optimize navigational pathfinding. These strategies include optimizing the search algorithm [Higgins02b], caching results of previous searches, and distributing plan formulation over several updates. Context preconditions can also be used to truncate searches that are heading down unfruitful paths.

# The Need for GOAP

With each new game released, the bar is set higher for AI behavior. As expectations for the complexity of character behavior rises, we need to look toward more structured, formalized solutions to creating scalable, maintainable, and re-usable decision-making systems. Goal-oriented action planning is one such solution. By letting go of the reins and allowing games to formulate plans at runtime, we are handing over key decisions to those who are in the best position to make them; the characters themselves.

# References


[AIPS02] AIPS'02, *Conference on AI Planning & Scheduling*, information about AIPS 2002 available online at www.laas.fr/aips/, 2002.

[Fu03] Fu, Dan, Houlette, Ryan, "The Ultimate Guide to FSMs in Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Higgins02a] Higgins, Dan, "Generic A* Pathfinding," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Higgins02b] Higgins, Dan, "How to Achieve Lightning-Fast A*," *AI Game Programming Wisdom*, Charles River Media, 2002.

[IPC02] 2002 IPC, *International Planning Competition*, information about IPC 2002 available online at www.dur.ac.uk/d.p.long/competition.html, 2002.

[McDermott98] McDermott, Drew, *Planning Domain Description Language*, information about PDDL2.1 available online at www.dur.ac.uk/d.p.long/IPC/pddl.html, 1998.

[Monolith02] *No One Lives Forever 2: A Spy in H.A.R.M.'s Way*, Monolith Productions/Sierra Entertainment, Inc., 2002. Toolkit and SDK available at nolf2.sierra.com/.

[Nilsson98] Nilsson, Nils J., *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, 1998.

[Russell95] Russell, Stuart, Norvig, Peter, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Inc., 1995.