The SimpleFPS Planning Domain: A PDDL Benchmark for Proactive NPCs

Stavros Vassos and **Michail Papakonstantinou**
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens 15784, Greece
{stavrosv,sdi0600151}di.uoa.gr

Abstract

In this paper we focus on proactive behavior for non-player characters (NPCs) in the first-person shooter (FPS) genre of video games based on goal-oriented planning. Some recent approaches for applying real-time planning in commercial video games show that the existing hardware is starting to follow up on the computing resources needed for such techniques to work well. Nonetheless, it is not clear under which conditions real-time efficiency can be guaranteed. In this paper we give a precise specification of SimpleFPS, a STRIPS planning domain expressed in PDDL that captures some basic planning tasks that may be useful in a first-person shooter video game. This is intended to work as a first step towards quantifying the performance of different planning techniques that may be used in real-time to guide the behavior of NPCs. We present a simple tool we developed for generating random planning problem instances in PDDL with user defined properties, and show some preliminary results based on SimpleFPS instances that vary in the size of the domain and two well-known planners from the planning community.

Introduction

In the field of artificial intelligence (AI), *proactive behavior* for an agent means "thinking before acting." Typically this amounts to the agent deliberating about the potential outcomes of her actions in order to choose the best available action or a sequence of actions that may achieve a desired goal. This is better understood in contrast to *reactive behavior* where no deliberation about the potential future state of affairs takes place.

The video game industry has been advertising the strong artificial intelligence skills of the non-player characters (NPCs) in the games. The characters are presented as being able to out-smart the player by sneaking, hiding, and following his moves. Typically though, the characters feature no deliberation capabilities and their behavior is not proactive in the previous sense. In fact, the characters are mostly reactive as in the vast majority of video games the high-level behavior of the characters is limited to following well-prepared predefined recipes that game designers specify in production time. Essentially, game developers create an illusion of intelligence using a few programming tricks for specifying the behavior of NPCs without the need to rely on deliberation or other sophisticated techniques from academic AI (Schaeffer, Bulitko, and Buro 2008; Millington and Funge 2009).

Nonetheless, it seems that in the last few years the game industry has reached a point where more sophisticated techniques for NPC behavior are necessary. This has been acknowledged both by gamers who thirst for smarter opponents that give the perception of truly autonomous human-like entities with their own agendas and realistic acting and sensing capabilities (Funge 2004; Nareyek 2004), as well as game developers who seek a scalable, maintainable, and reusable decision-making framework for implementing NPCs as the complexity of the game-world increases (Orkin 2005). To that end, some techniques from the AI literature have been used for achieving a proactive behavior based on planning, such as the noted case of the commercial game "F.E.A.R." (Orkin 2006).

A prerequisite for achieving proactive behavior is the ability to specify the goals of the agent and her capabilities in terms of affecting her surroundings, as well as a mechanism for combining this information to produce a concrete course of action at any given time. In this paper we focus on proactive behavior for NPCs in the first-person shooter (FPS) genre of video games based on goal-oriented planning (Orkin 2002), and attempt a first step towards quantifying the performance of planners that may be used real-time in a video game to guide the behavior of NPCs.

Some recent approaches for applying planning in video games (such as the case of "F.E.A.R." that was mentioned earlier) show that the existing hardware is starting to follow up on the computing resources needed for such AI techniques to work well. Nonetheless, it is not clear under which conditions certain guarantees can be imposed for memory usage and real-time efficiency for the planners in a game. In particular, it is very hard to evaluate what resources may be needed for a planner in a game unless the planner is actually implemented in the game, which makes it difficult for game developers to invest on adopting such a technique.

In this paper we focus on the simple language of STRIPS (Fikes and Nilsson 1971) for specifying a planning domain and problem instances for the domain. We introduce SimpleFPS, a specification for a game world domain that

represents some basic facts and properties that NPCs in a FPS typically consider in order to decide upon a high-level course of action. Based on this game world specification we proceed to present a tool for randomly generating problem instances (essentially game levels) that vary in size according to user input. We use this tool to generate two collections of instances that vary in the size of the domain, and report on the performance of two well-known planners from the planning community, BlackBox (Kautz and Selman 1999) and FastForward (Hoffmann and Nebel 2001), on these problems. We conclude with the evaluation the results and a discussion on future work.

STRIPS planning and the language of PDDL

In the area of classical planning one is faced with the following task. Given i) a complete specification of the initial state of the world, ii) a set of actions schemas that describe how the world may change, and iii) a goal condition, one has to find a sequence of actions such that when applied one after the other in the initial state, they transform the state into one that satisfies the goal condition. In this paper we focus on a specific formalism for representing planning tasks that is called STRIPS (Fikes and Nilsson 1971).

In propositional STRIPS planning the representation of the initial state, the action schemas, and the goal condition is based on the notion of literals from propositional logic. For example, the positive literal

$$\textit{NPC-Holding}(medkit_{23})$$

may be used to represent that the NPC is holding a particular med-kit, and the negative literal

$$\neg\textit{NPC-Holding}(gun_{12})$$

to represent that the NPC is not holding a particular gun.

The initial state is specified as a set of *positive literals* of the previous form. This set is assumed to give a complete specification of the world based on a *closed-world assumption*. This means that all the positive literals in the set are assumed to be true, and for all other literals that no information is included in the set it is assumed that the negative version of the literal is true.

For example, if the set describing the initial state includes only one literal of the form *NPC-Holding*(x), in particular the literal *NPC-Holding*$(medkit_{23})$, then it is assumed that for all other objects in the world the negative literal $\neg$*NPC-Holding*(x) is also true. In other words, a set that includes only the positive literal *NPC-Holding*$(medkit_{23})$ specifies that in the initial state the NPC is holding exactly one object, i.e., the med-kit $medkit_{23}$.

The action schemas specify what are the available actions in the world as well as when can each action be executed in a state and how is the state affected after the performance of the action. For example an action schema may specify the the preconditions and effects of the pick-up actions that an NPC may perform. One way to formalize this may be the action schema *place-in-inventory*(x) with preconditions that require that the NPC be located near the item x and effects that add the corresponding literal *NPC-Holding*(x) in the set that describes the state.

The preconditions and effects of actions are also treated as sets of literals. In the case of preconditions a set of positive literals specify what conditions need to be true in order for the action to be executable. Note that this can be tested by simply checking if all literals in the set of preconditions for an action are included in the set that describes the state. Similarly for the effects of an action, a set of positive and negative literals specify how the state should be transformed when the action is performed: all the *positive literals* in the set of effects are *added in the set* describing the state and all *negative literals* are *removed*. We will see particular examples of this in the next section.

Finally, a goal condition is also a set of positive literals and the intuition is that the goal is satisfied in a state if all the literals listed in the goal condition are included in the set that describes the state. A solution then to a planning task is a sequence of actions such that if they are executed sequentially starting from the initial state, checking for corresponding preconditions, and applying the effects of each action one after the other, they lead to a state description that satisfies the goal condition.

In this paper we focus on a specific syntax for describing propositional STRIPS planning tasks following the Planning Domain Definition Language (PDDL) (Mcdermott et al. 1998). PDDL is a standard language for specifying planning tasks that is widely used in the planning community. In PDDL, the specification of the predicates and the action schemas is separated from the specification of the initial state and the goal condition. The first part is typically referred to as the planning domain and the second part as the planning problem. In this way one can define a number of planning problems for the same planning domain.

The syntax of PDDL follows the logical representation of propositional literals as described above but a prefix notation is used. In this way, the positive literal *NPC-Holding*(x) may be represented as `(npc-holding ?x)`, and the negative literal $\neg$*NPC-Holding*(x) may be represented as `(not(npc-holding ?x))`. Note that variables are denoted using a preceding question mark. Special predicates are used for the specification of the planning domain with the intuitive meaning, including: `(:predicates ...)` and `(:action ...)`. Similarly, `(:objects ...)`, `(:init ...)`, and `(:goal ...)` are used to specify planning problems.

The SimpleFPS domain

In this section we present the SimpleFPS planning domain that is intended to capture some basic planning tasks that may be useful in a first-person shooter video game. We decided to represent only some more high-level actions that an NPC may perform, thus leaving actions such as path-finding and motion control to be treated in a lower-level.

That being said, a level of the game is divided into areas (in a real video-game these could be rooms) which are connected through way-points (e.g. doors). Each area has a number of points of interest (POIs) of various sorts including items, way-points, as well as the human player as a special point of interest. We consider each of these points of interest to have a specific location in the area which is stored at

a lower level. In order for the NPC to interact with them he should get close and then perform the desired action based on the type of point of interest he is close to (e.g. move from area to area if the point of interest is a way-point or attack using a melee weapon if the point of interest is the player).

The SimpleFPS domain uses a number of predicates to represent the available functionality in the game, as explained next.

The predicates of the domain

The predicates of SimpleFPS essentially represent the changing properties of the domain. We will present them in groups based on their usage.

The following predicates represent facts related to the state of NPC:

- `(npc-at ?a)`: the NPC is currently at area `?a`,
- `(npc-close-to ?p)`: the NPC is close to the point of interest `?p`.
- `(npc-not-close-to-point)`: the NPC is not close to any point in the area.
- `(npc-covered)`: the NPC is covered.
- `(npc-uncovered)`: the NPC is not covered.
- `(npc-holding ?o)`: the NPC is holding (e.g. as in an inventory) the object `?o`.
- `(npc-injured)`: the NPC is injured.
- `(npc-full-health)` the NPC has full health.
- `(npc-aware)`: the NPC has made contact with the player and knows his/her location.
- `(npc-unaware)`: NPC has not encountered the player.

The following predicates represent facts related to the state of the human player:

- `(player-wounded)`: the NPC has inflicted damage to the player.
- `(player ?p)`: the point of interest `?p` is identified as the human player.

The following predicates represent facts related to areas in the game world and their characteristics:

- `(connected ?area1 ?area2 ?waypoint)`: `?area1` is connected to `?area2` through `?waypoint`.
- `(waypoint ?w)`: `?w` is a way-point.
- `(lighted ?area)`: `?area` is lighted (so the NPC can see the points of interest inside).
- `(dark ?area)`: `?area` is dark.

The following predicates represent facts related to points of interest in the game world and their characteristics:

- `(point-of-interest ?p ?area)`: `?p` is a point of interest in `?area`.
- `(control-box ?p)`: `?p` is a control box (for turning on and off the lights).
- `(cover-point ?p)`: `?p` is a point where the NPC can take cover.
- `(item ?point)`: `?point` is an item.

Properties related to the items of the world are further described using the following predicates:

- `(keycard ?item ?waypoint)`: `?item` is a keycard that opens `?waypoint`.
- `(medikit ?m)`: `?m` is a med-kit that restores health.
- `(knife ?k)`: `?k` is a knife.
- `(gun ?g)`: `?g` is a gun.
- `(loaded ?gun)`: `?gun` is loaded
- `(unloaded ?gun)`: `?gun` is empty.
- `(ammo ?item ?gun)`: `?item` is ammo for gun `?gun`.
- `(has-nightvision ?gun)`: `?gun` has night-vision, therefore can be used to attack the human the player in dark areas.

The intuition is that the initial state of the game world (as well as any state that results from it) is represented as a set of positive ground literals of the kinds we have just described. We now proceed to present the action schemas in SimpleFPS that characterize the available actions for the NPC.

The available actions in the domain

Action schemas in PDDL (as in STRIPS) are defined in terms of their preconditions and effects using the available predicates of the domain. For example, the following PDDL statement is an action schema as it appears in the SimpleFPS domain that represents the actions of the NPC picking up an item of the game world.

```
(:action place-in-inventory
:parameters (?area ?item)
:precondition (and
  (npc-at ?area)
  (point-of-interest ?item ?area)
  (npc-close-to ?item)
  (item ?item)
  (npc-uncovered)
  (lighted ?area)
  )
:effect (and
  (not (point-of-interest ?item ?area))
  (npc-holding ?item)
  (not (npc-close-to ?item))
  (npc-not-close-to-point)
  )
)
```

The intuition with the action schema for `(place-in-inventory ?area ?item)` is that the action can only be performed when the NPC is close to a point of interest located in a area with light and the NPC is not in a cover state. The result of performing the action then is that the item is no longer located in the game world, therefore the corresponding literals are removed from the description of the current state, and a new positive literal is added in the state: `(npc-holding ?item)`.

The action schemas of SimpleFPS that represent the actions of taking cover and performing two different type of attacks follow.

```
(:action take-cover
:parameters (?area ?point)
:precondition (and
  (npc-at ?area)
  (point-of-interest ?point ?area)
  (cover-point ?point)
  (npc-close-to ?point)
```

```
  (npc-uncovered)
  )
:effect (and
  (npc-covered)
  (not (npc-uncovered))
  )
)

(:action attack-melee
:parameters  (?area ?weapon ?point)
:precondition (and
  (npc-aware)
  (npc-at ?area)
  (point-of-interest ?point ?area)
  (player ?point)
  (lighted ?area)
  (npc-close-to ?point)
  (npc-uncovered)
  (npc-holding ?weapon)
  (knife ?weapon)
  )
:effect (player-wounded)
)

(:action attack-ranged
:parameters  (?area ?weapon ?point)
:precondition (and
  (npc-aware)
  (npc-at ?area)
  (point-of-interest ?point ?area)
  (player ?point)
  (lighted ?area)
  (gun ?weapon)
  (npc-holding ?weapon)
  (loaded ?weapon)
  )
:effect (and
  (player-wounded)
  (unloaded ?weapon)
  )
)
```

Note that the preconditions for melee and ranged attack are different. In the first case the NPC needs to be close to the player, while in the second case the NPC need not be close to the player but a different kind of weapon is needed. The representation is very simplistic as for instance a knife could also be used in a ranged attack by throwing it towards the player, etc. Nonetheless, for SimpleFPS the point was to illustrate that different types of attack can be expressed using weapons with different capabilities.

We now give a complete list of the action schemas that we have specified in SimpleFPS along with a very brief description of their effects in the game world.

- `moving-to-patrol`: the NPC moves from the area it is located to another through the connecting way-point provided that it has not spotted the player.
- `moving-to-take-position`: the same but assuming that the NPC has spotted the player.
- `move-to-point`: the NPC moves close to a point of interest.
- `move-away-from-point`: the NPC moves away from a point of interest.
- `move-to-point-from-point`: the NPC moves from one point of interest to another within the same area.
- `make-accessible`: the NPC makes a way-point accessible by using the appropriate keycard.
- `make-contact`: the NPC spots the player.
- `take-cover`: the NPC takes cover at a respective point of interest.
- `uncover`: the NPC moves away from a cover point.
- `place-in-inventory`: the NPC places an item in his inventory.
- `reload`: reloads a gun using the correct kind of ammo.
- `attack-melee`: the NPC attacks the player using a melee weapon.
- `attack-ranged`: the NPC attacks the player using a ranged weapon.
- `sneak-kill`: the NPC attacks the player using a weapon with night vision in a dark area.
- `use-medikit`: NPC restores his health using a med-kit.
- `turn-on-lights`: NPC turns on the lights at the area it is located.
- `turn-off-lights`: NPC turns off the lights at the area it is located.

The parameters for each action schema as well as the preconditions and effects can be found in the full specification of SimpleFPS that can be found at `http://stavros.lostre.org/sFPS/`.

Problem instances in the SimpleFPS domain

In this section we present a simple tool we developed that can be used to generate random instances of planning tasks in PDDL based on the SimpleFPS domain we described in the previous section. Given a number parameters the tool creates a PDDL problem for the SimpleFPS that specifies the initial state of the game world and a goal condition that should be satisfied. The parameters are specified as command-line arguments as follows:

- `-a <int>`: the number of areas of the generated level.
- `-c <float>`: the probability that two areas are connected through a way-point.
- `-n <int>`: the total number of points of interest in the level. Each one can either be a gun, a knife, a first-aid kit or a cover point.
- `-g <goal condition>`: this specifies the goal condition that will be specified in the planning problem; `<goal condition>` can be one of the following:
  - `g1`: the goal is that the NPC inflicts damage to the player.
  - `g2`: the goal is that the NPC takes cover.
  - `g3`: the goal is that the NPC restores its health to full.
  - `g4`: the goal is the conjunction of the three goals above.
- `-l <int>`: the number of different levels to be generated.

Each time a level is generated some statistics are printed such as the starting area of the NPC and the player, the type of each item placed in the level, the number of way-points and key-cards. Also, the file name of the generated level includes the value for each parameter for easier inspection.

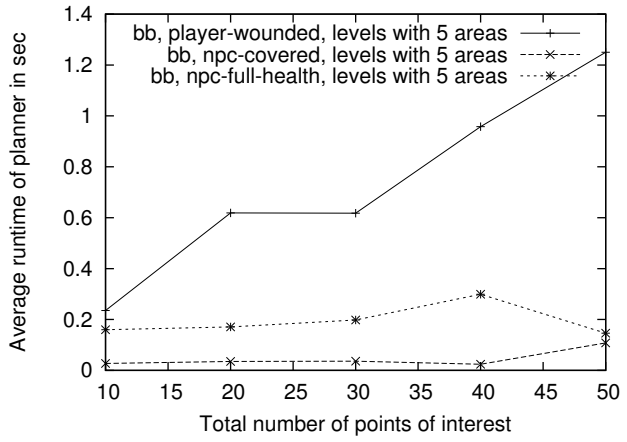

Figure 1: Performance of BlackBox on SimpleFPS problems from the collection `sFPS-a5-c0.7`

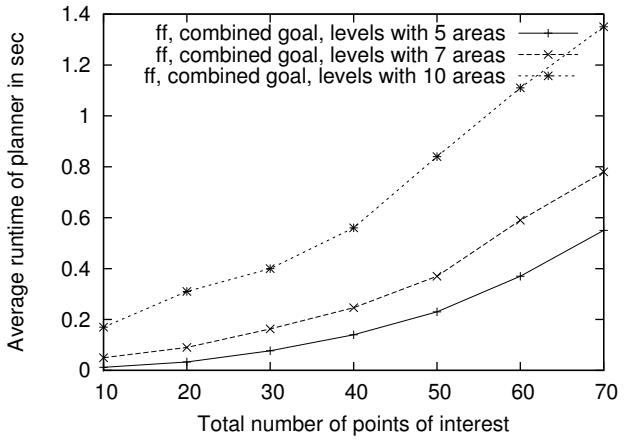

Figure 2: Performance of FastForward on SimpleFPS problems from all three collections `sFPS-a5-c0.7`, `sFPS-a7-c0.7`, and `sFPS-a10-c0.7`

The rest of the details for each generated level are specified randomly using some fixed probabilities in order to decide, e.g., whether a particular way-point is open or not, and a gun is loaded or not. A few hard-coded rules ensure that the generated levels are intuitive, e.g., every time a closed way-point is inserted, a key-card is also placed at a random area. Similarly, for every empty gun that is generated, some ammo for that gun is also placed at a random area.

Finally, we note that the source code of this tool can be found at the link provided earlier.

Three collections of SimpleFPS planning tasks and some preliminary results

As the main motivation for SimpleFPS is the evaluation of planning techniques in video game worlds, we now present some preliminary results based on two well-known planners from the planning community that rely on different techniques for finding a solution, namely BlackBox (Kautz and Selman 1999) and FastForward (Hoffmann and Nebel 2001). The intention was to get a first idea about how difficult this problem is for existing planners and maybe some intuition as for which techniques are more suited for this kind of domains. At the same time we wanted to provide a set of problems that could act as a benchmark for other people interested to apply planning techniques in video games with similar characteristics.

Using the tool we developed for SimpleFPS, we generated three collections of planning problems with a different number of total areas: `sFPS-a5-c0.7`, `sFPS-a7-c0.7`, and `sFPS-a10-c0.7`. The first collection includes problems with 5 areas such that the probability that two areas are connected is 0.7, and a number of points of interest that range from 10 to 100. In particular, for each n in $\{10, 20, \ldots, 100\}$, we generated 10 problem instances with n points of interest and four different versions of each instance that correspond to the four goal conditions `g1`, `g2`, `g3`, and the combined goal `g4`. Similarly for the other two collections problems with 7 and 10 areas where generated. The collections can be found at the link provided earlier.

Figure 1 shows the performance of BlackBox for problems in collection `sFPS-a5-c0.7`. Unfortunately, the planner had trouble solving problems with more than 50 points of interest, often taking more than a minute to respond (which is clearly not desirable for an FPS game). In fact, the combined goal of achieving `g1`, `g2`, and `g3` at the same time needs more than two minutes even for instances with 5 areas and 20-40 total points of interest. Therefore we only show the running-time of the planner for small instances with up to 50 items, and only for each of the simple goal conditions `(player-wounded)` (`g1`), `(npc-covered)` (`g2`), and `(npc-full-health)` (`g3`) separately, for which the planner responds in less than 1.5 seconds.

Figure 2 shows the performance of FastForward for problems in all three collections `sFPS-a5-c0.7`, `sFPS-a7-c0.7`, and `sFPS-a10-c0.7`. FastForward was able to perform better in the sense that it can handle much larger problems for similar running time. Unlike BlackBox, FastForward was able to respond within 3 seconds for all problem instances. Figure 2 shows the running-time of the planner for instances with up to 70 items for the combined goal `(and player-wounded npc-covered npc-full-health)`, for which the planner responds in less than 1.5 seconds. Note also that we only report the running-time of the planner for the most difficult of the goal conditions, that is, the combined goal, for which BlackBox was unable to respond at all within 3 seconds for any instance. Observe also that for problems small in size, e.g., up to 50 points of interest and up to 10 areas, FastForward achieves a sub-second performance.

Discussion and future work

In this section we briefly discuss the preliminary results presented in the previous section and some directions for future work that we intend to investigate.

As far as the running time of BlackBox and FastForward

is concerned, clearly, in order for real-time planning to be practical in commercial video games the reported numbers need to improve by one or two orders of magnitude, so that such planning problems can be solved up to many times per frame in the game. Nonetheless, it should be noted that the reported results are only a shallow evaluation using some of the existing techniques. There are a few reasons to believe that much better performance can be achieved.

For one, the planners developed in the planning community aim for completeness (and often optimal solutions) which is a very demanding task. For a video game setting it may make more sense to search for sub-optimal or approximate solutions that can be computed very fast. This requires a close collaboration between AI academics and game developers so that the specific needs of game developers can be specified and quantified.

Moreover, even the simple planning problems we consider in SimpleFPS may require the planner to look into solutions with length up to more than ten actions. Apart from the fact that this is computationally demanding, in most FPS video games a plan that consists of more than a few actions is of little use as the state of the game world changes very fast and large plans become obsolete very quickly. Instead of actually obtaining a plan of fifteen actions, an NPC may better off searching for a plan of at most five actions and if such a plan cannot be found, revert back to using some predefined plan or strategy. Searching for plans of bounded length can dramatically decrease the time needed to either find a solution or return with failure. In this sense, the preliminary results reported in the previous section can be thought as actually having a positive flavor.

Similarly, we would also like to investigate better ways that planning could be used to specify the behavior of NPCs, not necessarily bound to searching for a solution as a sequence of actions and strictly following the solution found. In fact, we used the term "proactive behavior" in order to stress that there is a wider class of behaviors that can be achieved based on the same principles. To that end we intend to investigate languages that combine planning as well as a way to control or fine tune the outcome of the planning mechanism in the spirit of the agent programming language Golog (Levesque et al. 1997) and IndiGolog (De Giacomo et al. 2009). In these languages one can define high-level agent programs that can be seen as abstract plans that show how the intended behavior for the agent should look like. In the execution of such a program planning is used implicitly to achieve some of the requirements specified at certain parts of the program, but only at the time that it is needed. A few approaches toward this direction include (Jacobs, Ferrein, and Lakemeyer 2005) and (Ferrein 2010).

Finally, we note that to the best of our knowledge SimpleFPS is the first approach for building a benchmark for evaluating different planning techniques in the genre of FPS video games. The PDDL language has been used before for performing real-time planning in video game worlds, e.g., (Bartheye and Jacopin 2009), but not with the intention of providing a benchmark. It should also be noted that SimpleFPS is based on the approach of the game F.E.A.R. as it was described in (Orkin 2006).

References

Bartheye, O., and Jacopin, E. 2009. A Real-Time PDDL-based planning component for video games. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. The AAAI Press.

De Giacomo, G.; Lespérance, Y.; Levesque, H. J.; and Sardina, S. 2009. IndiGolog: A High-Level programming language for embedded reasoning agents. 31–72.

Ferrein, A. 2010. golog.lua: Towards a Non-Prolog Implementation of Golog for Embedded Systems. In Hoffmann, G., ed., *Proc. of AAAI Spring Symposium 2010 on Embedded Reasoning*, (SS-10-04).

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Funge, J. D. 2004. *Artificial Intelligence For Computer Games: An Introduction*. MA, USA: A. K. Peters, Ltd.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Jacobs, S.; Ferrein, A.; and Lakemeyer, G. 2005. Controlling unreal tournament 2004 bots with the logic-based action language GOLOG. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE05)*. CA, USA: Morgan Kaufmann publishers Inc.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th international joint conference on Artificial intelligence*, 318–325. CA, USA: Morgan Kaufmann Publishers Inc.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–83.

Mcdermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games, Second Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2nd edition.

Nareyek, A. 2004. Artificial intelligence in computer games - State of the art and future directions. *ACM Queue* 10(1):58–65.

Orkin, J. 2002. Applying goal oriented action planning in games. In *AI Game Programming Wisdom 2*. Charles River Media. 217–229.

Orkin, J. 2005. Agent architecture considerations for Real-Time planning in games. In *Artificial Intelligence & Interactive Digital Entertainment (AIIDE)*.

Orkin, J. 2006. Three states and a plan: The AI of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC)*.

Schaeffer, J.; Bulitko, V.; and Buro, M. 2008. Bots get smart. *IEEE Spectrum* 45(12):44–49.