iThink: A Library for Classical Planning in Video-Games

Vassileios-Marios Anastassiou, Panagiotis Diamantopoulos, Stavros Vassos, and Manolis Koubarakis

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens 15784, Greece
{astyanax,panos_10d,stavrosv,koubarak}@di.uoa.gr

Abstract. Academic artificial intelligence (AI) techniques have recently started to play a more central role in the development of commercial video games. In particular, classical planning methods for specifying a goal-oriented behavior have proven to be useful to game developers in an increasingly number of cases. Motivated by the fact that there is no clear standard for developing a goal-oriented behavior in video games, we present iThink, a framework that allows the use of academic techniques for classical planning in order to achieve goal-oriented behavior in a real game developing environment. In our work we focus on STRIPS, a well-studied framework for classical planning, and Unity3D, a popular game engine that is becoming an emerging standard for, so-called, "indie" game development. Except for being a useful tool for game developers, we believe that iThink can be used in education, providing a modern and fun environment for learning and experimenting with classical planning.

1 Introduction

Traditionally, the artificial intelligence (AI) methods used in video games have not relied on techniques that have been developed in the academic field of AI. Essentially, game developers create an illusion of intelligence using a few programming tricks without the need to rely on deliberation or other sophisticated techniques from academic AI [6,13]. As a fast and robust implementation is very critical for the video game industry, this approach proved to be effective and convenient. In particular, this meant that for most video games the behavior of non-player characters (NPCs) involved a number of pre-programmed responses that vary depending on conditions about the game-world and the player.

This was not a problem for many years as the video game industry was able to evolve remarkably well based on the advances in other areas such as computer graphics whose impact on the game-play was celebrated by the players. The design of more challenging NPCs was typically counter-weighed by giving them more power than the human players (e.g., extra strength or speed, omniscience). 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

I. Maglogiannis, V. Plagianakos, and I. Vlahavas (Eds.): SETN 2012, LNAI 7297, pp. 106–113, 2012.

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 [4,7], 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 [9].

A notable academic AI technique that proved to be useful for this purpose is *classical planning*, which is often referred to in video game industry as *goal-oriented action planning (GOAP)*. The main idea of this technique is to provide an NPC with a set of *actions*, a description of the *effects* of these actions in the game-world, and a *goal* the NPC should try to achieve by using the available actions. This approach has received much attention in the video game industry mainly due to the noted case of the commercial game "F.E.A.R." [8] that used a simplified version of STRIPS planning [3] for NPC behavior.

The implementation of STRIPS planning in the game of "F.E.A.R" relies on important restrictions and optimizations in order to achieve real-time performance. Similarly, in subsequent games that planning has been used, the implementation is game-specific and cannot easily be reused in other games. On the other hand, existing academic frameworks for classical planning that aim for generality, such as planners that conform to the Planning Domain Definition Language (PDDL) [5], cannot be directly used in a game engine: apart from performance issues that cannot be easily overcome (e.g., academic planners typically assume no restrictions in memory resources), a major problem is connecting the *"real"* objects in a game-world with their corresponding *"symbolic"* ones in the underlying logic-based representation used for planning.

In this paper we address the practical problem of providing an planning framework that can be used in a wide game development environment. We report on our work on developing iThink, a STRIPS planning framework that aims for generality, ease of use, and is integrated in a real game engine. iThink is targeted to Unity3D, a popular game engine that is becoming an emerging standard for, so-called, "indie" game development but is also used in "AAA" titles, that is, commercial games of the highest quality. Except for being a useful tool for game developers, we believe that iThink can be used in education, providing a modern and fun environment for learning and experimenting with classical planning.

The rest of the paper is organized as follows. In Section 2 we go over the basic details of STRIPS planning. In Section 3 we present the iThink framework that implements STRIPS planning in the game engine Unity3D. In Sections 4 and 5 we discuss related and future work, and in Section 6 we draw some conclusions.

2 STRIPS Planning

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 *action 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 work we focus on the STRIPS formalism for representing planning tasks [3]. In STRIPS, the representation of the initial state, the action schemas, and the goal condition is based on literals from predicate logic. For example, in a video game scenario the positive literal $Gun(o)$ may be used to represent that o is a gun, and $Holding(knife_{23})$ to represent the fact that the NPC is holding a particular knife. Similarly, the negative literal $\neg Holding(o)$ may be used to represent that the NPC is not holding the gun o.

The *initial state* is specified as a set of positive literals. This set provides a complete specification of the state based on a *closed-world assumption*. That is, for all ground literals not included in the set, it is assumed that the negative version of the literal is true. The *actions* then of the domain affect the current state by means of adding and deleting literals. For example, $pick\text{-}up(o, room_1)$ may be used to represent the action of the NPC picking up the gun o that is located in $room_1$, which affects the current state by adding the literal $Holding(o)$.

The *action schemas* are the general rules that specify the available ground actions in the domain. For $pick\text{-}up(o, room_1)$, the corresponding action schema may look like $pick\text{-}up(x_1, x_2)$, followed by two sets of literals that specify the preconditions and the effects of any particular action that is a ground instance of this schema. The set of preconditions specifies what literals need to be present in a state in order for the action to be executable in that state. The set of effects specifies how the state should be transformed when the action is executed: positive literals are added in the state description, and negative literals are removed.

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 problem 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 the next section we will present iThink, a framework for STRIPS planning that is embedded in the Unity3D game developing environment.

3 The iThink Framework in Unity3D

Unity3D[1] is an integrated authoring tool that provides out-of-the-box functionality for building video-games and simulations. Unity3D implements a full-featured high-quality game engine, and provides tools for developing and managing content. The developers can code the functionality of the game-world using Javascript, C#, and Boo (a python-inspired .Net language), in a programming environment that is based on the FOSS Mono platform.

A free license supporting a basic feature set is available, providing easy access to any developer. Along with its potential portability across different systems, Unity3D is a promising ground for easy development and deployment of educational, research or commercial projects. The features of Unity3D along with

[1] `http://unity3d.com/`

the increasing popularity and the extensive documentation freely available online led us to choose Unity3D as the ideal game developing environment for implementing iThink.

3.1 iThink Overview

The development of iThink was done with two goals in mind. The first one concerns game developing and aims for ease of use and modularity for the programmers, while the second concerns academic research and aims for providing the ground for studying planning techniques in a real software development setting. In order to achieve these (sometimes contradictory) goals, we had to make a few compromises so as to serve both worlds.

iThink provides the necessary methods for specifying a STRIPS planning problem about elements of the game-world in Unity3D. As STRIPS is based on predicate logic, we had to come up with a convenient way of initializing logical literals that refer directly to objects in the software programming sense. For this purpose we utilized the tag functionality of Unity3D according to which any object of the game-world (including items, way-points, NPCs, and the human player) may be associated with a number of tags. Using the integrated developing environment of Unity3D, a developer can specify easily the types of available objects, which iThink can then use for planning purposes. For instance, for a particular object o that is tagged with the label "gun", the literal $Gun(o)$ will be added to the iThink knowledge base to be used for planning purposes.

The initial state of the planning problem is specified using literals that are built automatically using the specified tags, as well as others that are specified by the developer via appropriate methods. Similarly, the goal condition is specified by the developer as a set of literals. The tag functionality is also used in the specification of action schema strings in order to restrict the generated actions to those that make sense in the current state of the game-world. For example, the *pick-up*(x_1, x_2) action schema may only be used to instantiate actions where x_1 is an item and x_2 a location of the game-world.

As far as finding a solution to a planning problem is concerned, iThink provides methods for employing a forward-search method that works in the state-space in a breadth-first or depth-first manner for the uninformed case and a best-first manner when a heuristic is provided[12]. Note that iThink is general enough to allow the implementation of other available methods for classical planning as well. iThink classes are organized in a way that can be easily extended and adapted to meet the particular needs of the developer. Each iThink class implements an atomic building block for representing STRIPS planning problems in Unity3D and effectively searching for solutions. The most important classes of iThink are presented in the next section.The full implementation is available online at the project's webpage: `http://code.google.com/p/ithink-unity3d/`

3.2 Implementation Details of iThink

The core functionality of iThink is implemented in nine C# classes as follows.

```
public class SimpleFPSAgent : MonoBehaviour
{
    iThinkBrain brain;
    public string[] schemaList = {
        "ActionMove-3-Tag~loc-Tag~loc-Tag~dir",
        "ActionTurn-2-Tag~dir-Tag~dir",
        "ActionShoot-4-Tag~loc-Tag~loc-Tag~dir-Tag~gun",
        "ActionStab-2-Tag~loc-Tag~knife",
        "PickUp-2-Tag~knife-Tag~loc",
        "ActionPickUp-2-Tag~gun-Tag~loc"
    };
    public void Awake() //executed when NPC is constructed
    {
        brain = new iThinkBrain();

        List<String> tags = new List<String>();
        tags.Add("dir"); tags.Add("loc"); tags.Add("player");
        tags.Add("npc"); tags.Add("gun"); tags.Add("knife");
        brain.sensorySystem.OmniUpdate(this.gameObject,tags);

        brain.ActionManager = new iThinkActionManager();
        brain.ActionManager.initActionList(
            this.gameObject,
            schemaList,
            brain.getKnownObjects(),
            brain.getKnownFacts()
        );
    }
    public void Update() //executed at every frame of the game
    {
        //... code that specifies the initial state and goal

        brain.planner.forwardSearch( //invokes planner
            brain.startState,
            brain.goalState,
            brain.ActionManager,
            1); //specifies the search method
        brain.planner.getPlan().debugPrintPlan();

        //... code that uses the plan for NPC behavior
    }
}
```

Fig. 1. Example of iThink usage in a simplified game-world

- **iThinkFact** is used to specify STRIPS literals. An iThinkFact instance specifies the predicate name of the literal, a collection of game objects as the arguments of the literal, and the polarity of the literal (positive or negative).
- **iThinkState** is used to specify STRIPS states. An iThinkState instance is simply a set of literals, i.e., iThinkFacts instances.

- **iThinkAction** is used to specify ground actions. An iThinkAction instance specifies the name of the action, a set of preconditions as a set of positive literals, and a set of effects as a set of positive and negative literals.
- **iThinkActionSchema** is used to specify STRIPS action schemas. An instance of iThinkActionSchema specifies the name of the action schema, the number of arguments, and the type of objects that can be used as arguments when generating a ground action using the schema. This information is given to the constructor of the class using a properly formatted string.
- **iThinkActionManager** generates and stores a collection of all the available ground actions that can be generated using the information about the available action schemas and the iThink knowledge base of available objects.
- **iThinkSensorySystem** facilitates an automatic generation of literals that describe the current state of the game-world using Unity3D tags on objects.
- **iThinkPlan** is used to specify STRIPS plans, that is, solutions to a STRIPS planning problem. An instance of iThinkPlan specifies a sequence of actions, i.e. iThinkAction instances, and is also used to represent partial plans.
- **iThinkPlanner** provides search utilities and is responsible for the execution of the planning process. Currently, a forward-search method is implemented that works in a breadth-first or depth-first manner for uninformed search and in a best-first manner when a heuristic is provided.
- **iThinkBrain** is the main class that manages a planning problem using the other iThink classes. iThinkBrain can be the basis of specifying any planning problem that concerns objects of the game-world.

We proceed to show the usage of iThink with an example involving an NPC.

3.3 iThink Usage Example

An intended use of iThink is to associate an instance of iThinkBrain with an NPC. In this case, whenever the NPC needs to find a high-level plan for achieving a goal in the game-world, the implementation of the NPC can use the iThink functionality to update the initial state of the planning problem according to the current state of the game-world, specify a goal, and use the available planning methods to find a sequence of actions that achieves this goal.

In Figure 1 we show an example, omitting some details due to space limitations. The main tasks that the developer needs to do are the following.

1. Define an NPC class that contains an iThinkBrain object (lines 1–3).
2. Specify, as properly formatted strings, the action schemas that iThinkBrain will evaluate. The strings show the available actions and the types of objects of the game-world that can be used to initialize them (lines 4–11).
3. In the initialization of the NPC, use a sensor method of iThinkSensorySystem to identify the available objects of the game-world (lines 16–19), and initialize the iThinkActionManager that generates the available actions (lines 21–27).
4. In the method that updates the behavior of the NPC, inform iThinkBrain about the initial state and the desired goal condition to pursue (insert code at line 31), invoke the iThinkBrain planner (lines 33–38), and then use the result to specify the NPC behavior (insert code at line 40).

4 Related Work

There are many implementations of planning techniques in the literature. The novelty of iThink is that it departs from the strictly symbolic approach typically adopted in academic AI and aims for a planning capability that operates directly on the software programming objects. In order to embed the symbolic STRIPS approach in Unity3D in a way that is easy to inter-operate with the existing methods and tools of the developing environment, we had to deal with problems that are usually not considered in academic AI research. This is often no easy task and our approach succeeded in remaining faithful to the original STRIPS formalism, while providing a programming framework that expresses the logical background of STRIPS using familiar constructs such as tags and variables.

Similarly, there are a few approaches that attempt the incorporation of GOAP in video games. Orkin used a simplified STRIPS approach in the notable case of the game "F.E.A.R.' [8]. Bjarnolf investigated threat analysis using a GOAP system that worked in an "Observe, Orient, Decide, Act" manner [1]. Long compared a GOAP implementation with a finite-state machine implementations, suggesting that planning techniques are superior when considering both practicality and performance [2]. Peikidis investigated the application of GOAP techniques in a strategy game [10] and Pittman studied command hierarchies in a GOAP setting as a solution to code maintenance issues [11].

All these approaches follow closely the architectural decisions of F.E.A.R in order to achieve real-time efficiency. Nonetheless, these approaches are based on strong simplifications of the STRIPS paradigm that drastically reduce the expressiveness of the approach. Moreover, each approach focuses on specific game constructs and details, essentially making it appropriate only for the particular application case. In contrast, iThink aims for generality and modularity. By being faithful to the STRIPS logical background, iThink provides a powerful framework for representing different planning environments in Unity3D.

5 Future Work and Extensions

Our ultimate goal is to evolve iThink into an efficient AI module that can be used in the development of commercial video games. For our future work we intend to implement and experiment with heuristic planning methods that are specifically tailored for particular video game genres, such as for example the first-person shooter (FPS) genre. We also want to investigate how iThink may be extended to account for planning that involves multiple agents using a blackboard architecture and the possibility of taking advantage of multi-threading for achieving better performance and response times.

There are also some parts of iThink that we want to improve, such as for example the way the action schemas are specified in the current version. Our intention is to fully integrate iThink in the Unity3D editor so that the developer can specify such information using a visual interface that will further simplify the modeling of the underlying planning domain. Along these lines, we also plan on building debugging tools for testing and visualizing the planning process.

6 Conclusions

Classical planning has proven to be useful for achieving a dynamic, emergent behavior for NPCs in video games. Our approach is a novel way of employing academic techniques for classical planning in a real video game developing environment. iThink aims for generality and modularity, providing the means for specifying a wide range of planning problems according to the particular needs of game developing. Except for being a useful tool for game developers, we believe that iThink can be used in education, providing a modern and fun environment for learning and experimenting with classical planning.

References

1. Bjarnolf, P.: Threat analysis using goal-oriented action planning. B.Sc. Thesis, University of Skovde, School of Humanities and Informatics (2008)
2. Edmund, L.: Enhanced NPC Behaviour using Goal Oriented Action Planning. Master's thesis, University of Abertay Dundee, School of Computing and Advanced Technologies, Division of Software Engineering (2007)
3. Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, 189–208 (1971)
4. Funge, J.D.: Artificial Intelligence For Computer Games: An Introduction. A.K. Peters, Ltd., MA (2004)
5. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL - the planning domain definition language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Comp. Vision and Control (1998)
6. Millington, I., Funge, J.: Artificial Intelligence for Games, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2009)
7. Nareyek, A.: Artificial intelligence in computer games - State of the art and future directions. ACM Queue 10(1), 58–65 (2004)
8. Orkin, J.: Three states and a plan: The AI of F.E.A.R. In: Proceedings of the Game Developer's Conference, GDC (2006)
9. Orkin, J.: Agent architecture considerations for Real-Time planning in games. In: Artificial Intelligence & Interactive Digital Entertainment, AIIDE (2005)
10. Peikidis, P.: Demonstrating the use of planning in a video game. B.Sc. Thesis, University of Sheffield, CITY Liberal Studies, Dept. of Computer Science (2010)
11. Pittman, D.L.: Enhanced NPC Behaviour using Goal Oriented Action Planning. Master's thesis, University of Nebraska-Lincoln (2007)
12. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall (2002)
13. Schaeffer, J., Bulitko, V., Buro, M.: Bots get smart. IEEE Spectrum 45(12), 44–49 (2008)