Planning with Hierarchical Task Networks in Video Games

John-Paul Kelly
Department of Engineering
Australian National University
Canberra, ACT

Adi Botea
NICTA and
Australian National University
Canberra, ACT

Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA

Abstract

Artificial intelligence (AI) technology can have a dramatic impact on the quality of a video game. AI planning methods are useful in a wide range of game components, including modules to control the behaviour of fully autonomous units. However, planning is computationally expensive and the CPU and memory resources available at runtime to a game AI module are scarce. Offline planning can be a good strategy to avoid a runtime performance bottleneck.

In this work we apply planning with hierarchical task networks (HTNs) to video games. HTNs can speed up planning dramatically, since search is guided with human-encoded knowledge. We describe an architecture that computes plans offline. This can be seen as a form of generating scripts automatically, replacing the traditional approach of composing them by hand. The results are very encouraging. Scripts are automatically generated at a level of complexity that would require a great human effort to create.

Introduction

Artificial intelligence can have a major impact on the quality of a video game. AI techniques such as planning and search can make the behavior of fully autonomous characters that populate a game look intelligent.

Planning is computationally expensive. This issue is of particular importance in games, where solutions have to be available in real time and the CPU and memory resources are limited. In this article we apply planning with hierarchical task networks (HTNs) (Sacerdoti 1975; Tate 1977) to the domain of commercial computer games. HTNs are hand-coded structures that encode knowledge about the domain at hand. They show how abstract tasks (e.g., eat) can be decomposed into a sequence of more concrete tasks (e.g., go to restaurant, order food, enjoy food). Several decomposition methods can exist for a given task. For example, it is also possible to buy ingredients, cook, and eat at home. Searching with HTNs explores only paths that fit into the HTN structure, reducing the overall planning effort considerably. We focus on offline planning, effectively avoiding runtime performance bottlenecks.

Non-playing characters (NPCs) are an example of entities under the control of the computer. Traditionally, NPCs were used with the only goal of populating the background of the game world. Their behavior was limited to basic animations, with little or no interaction with the rest of the world. In contrast, intelligent NPCs could act according to a meaningful plan and interact with each other.

As a motivating example for our work, consider the problem of modelling the behavior of NPCs. A traditional approach is to use *scripts*, simple plans that are precomputed by hand and cached to be used at runtime. More recently, runtime planning modules have been added to game engines (Sierra 2007). Each of these have their advantages and shortcomings, and our goal is to combine the strengths of both.

A significant advantage of scripts is that they can be used at runtime with little CPU overhead, since their creation is performed offline. On the other hand, scripts require lots of human effort to generate. The process is error-prone and scripts are typically limited in size. Many scripts need to be generated to cover a reasonable range of situations that might occur in a game. Runtime automated planning can compute solutions as needed, taking into account the current state of the game and the objectives to achieve. The price is a potentially significant computational effort.

HTNs combine the advantages of both scripting and planning. HTN planning is fast, since search is tightly guided with human-encoded knowledge. In this work, HTNs and scrips are related in two ways. Firstly, a hierarchical task network can be seen as an enhanced form of a script. Instead of writing a complete solution, details are abstracted away and a more general structure, which corresponds to many problems, is cached. When a new problem needs to be solved, computing a complete solution reuses the HTN and adds on top of it the details specific to that instance. Secondly, plans computed with HTN planning are represented as scripts by translating them from the planning format into the scripting language. In this way, plans can be plugged into the game with no need to add code to the game product.

One hierarchical task network can model the behaviour of many types of characters, which may have high-level similarities but differ significantly in the concrete ways they act. Consider a scenario with creatures getting hungrier over time. All hungry NPCs try to feed themselves, but how exactly this happens differs from one character type to another. Some human-like creatures buy food, some others go hunting. Vampires suck the blood of their prey. The

HTN abstract task of getting food is shared by all characters. Lower-level tasks are more specialized. Adding a new NPC type requires a straightforward refinement of the HTN with new specialized tasks, methods and actions. An HTN with more lower-level methods does not necessarily imply a larger search effort. Simple preconditions that check the type of a character can prevent the planner from trying task decompositions that are irrelevant for the character at hand.

In many modern games, users can enhance a game with new elements such as graphical environments, characters, and scripts that model their behaviour. Since HTN planning needs input from expert users, an interesting question is whether the game development community for user created content, which is very large and heterogeneous in terms of programming skills, would welcome such an addition to off-the-shelf game products. We argue that HTNs would make it into a standard game feature rather easily. Scripting has the merit of creating a precedent in getting users accustomed to express their input in a structured language. Moving up to HTNs could be done with minimal user effort. If desired, graphical interfaces can be designed to assist in generating HTNs.

HTN planning can be applied to several classes of video games, including real-time strategy (RTS), role playing (RPG), and first person shooter (FPS) games. Specific implementations can result in either offline or online planning modules. As already mentioned, this paper focuses on offline planning, which can be seen as an automated way of generating scripts, scaling up to more complex scripts and reducing the human effort.

The rest of this paper is structured as follows: We continue with related work. The next section describes our planning approach in detail. Experimental evaluation is the topic of the second last section. The last part contains conclusions and future work ideas.

Related Work

This section places our work into a proper context by reviewing related work from both academia and industry. We start with a short survey of HTN planning then move on to a review of controlling NPC behaviour in video games. The idea of hierarchical task networks dates back to the work of (Sacerdoti 1975) and (Tate 1977). SHOP2 by Nau *et al.* (2003) is a successful modern planner that implements HTNs. The basic version works for classical planning but extensions for nondeterministic planning (Kuter and Nau 2004) and probabilistic planning (Kuter and Nau 2005) have been introduced. Work on automatically learning parts of an HTN such as method preconditions is reported by Ilghami *et al.* (2002). Wilkins and desJardins (2001) advocate the need for HTNs and, more generally, *knowledge-based planning* in complex realistic applications. Real-life problems where HTN planning has been successful include production line scheduling (Wilkins 1988) and the game of bridge (Smith *et al.* 1998).

In games, scripting and finite state machines are traditional approaches to controlling NPC behaviour. More recently, the games industry has adopted methods based on AI planning and scheduling. Scripting has been used in games ranging from RPGs such as Bioware's NEVERWINTER NIGHTS to FPSs such as Epic Games' UNREAL TOURNAMENT. Scripts can be used to give NPCs simple behaviours. They are written offline in a high-level, game-specific language. Scripting is relatively simple even to people with minimal traditional programming experience. Many NPC behaviours can be scripted, including conversation trees or general social characteristics. Scripts typically execute sequentially in an infinite loop with the only concept of state built through using local variables. When modelling more elaborated scenarios, scripts quickly become quite long and complex.

Steps have been taken towards automating script generation (McNaughton *et al.* 2004). SCRIPTEASE is an attempt at simplifying the process of writing scripts through simple pattern templates. Relatively complicated behaviours can be created with a GUI without the need for explicitly programming them. While this can reduce the development time, it still inherits many of the complexities that come with developing scripts. The developer must still manually choose what behaviours to initiate and when for each character, which can quickly become very complicated as the number of characters and complexity of the world increases.

At the moment, the use of state machines is a lot more common than the use of planning for controlling NPC behaviour. Games that have implemented state machines range from FPSs such as ID Sofware's QUAKE series to RTSs such as Blizzard Software's WARCRAFT III. Some recent games such as Bungie's HALO 2 use hierarchical state machines (Isla 2005). The use of state machines is regarded as less feasible for the long term and is even proving difficult in current generation games (Orkin 2003). A major problem is that the complexity of state machines grows quickly when the behaviour of a character becomes slightly more sophisticated.

Different planning approaches, including simple forms of hierarchical planning have been used to control the behaviour of NPCs (Munoz-Avila and Fisher 2004). This is typically used to control opponents in First Person Shooter style games such as Monolith's F.E.A.R. (Sierra 2007) or Epic Games' UNREAL TOURNAMENT (Games 2007). In the past, having simple behaviours such as moving between ammo caches and chasing any opponents they encounter was acceptable NPC behaviour in FPSs. In more modern games, NPCs are expected to be able to coordinate as squads and perform advanced tactics such as sending for backup or flanking (Orkin 2006).

Another approach to controlling NPCs is scheduling as seen in Bethesda's THE ELDER SCROLLS IV: OBLIVION. Each NPC has a series of tasks with scheduled times and preconditions. For example, each day an NPC might go for lunch at 12pm and then go to work for 4 hours if it is a weekday (Stein 2007). This can be used in conjunction with scripts to control more detailed or character specific actions. This can be an effective approach but it is not without its downsides. Manual scheduling is needed to integrate new characters into the game world. For example, if a character needs to buy something from a store, then the keeper of the store needs to be there. If there are a few characters and we

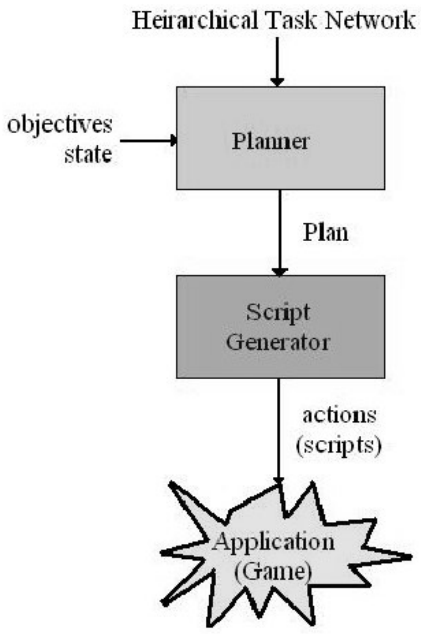

Figure 1: Architecture for offline planning.

want to change the behaviour of any one of them, then we will need to look at the schedules of all of the other characters to make sure this does not cause any conflicts. As more characters with more complicated schedules are introduced, this can become unfeasible.

HTN Planning in Games

The first part of this section presents an approach for automatic script generation with offline HTN planning. An example comes next. The following part contains a few insights to be considered when planning is used in games. Finally, we briefly compare our work with SCRIPTEASE, another approach for automatic script generation in games.

Offline Planning

Accepting user input in the form of scripts, a standard feature of many modern games, makes the implementation of offline planning relatively easy and natural, saving lots of development effort. The offline planning architecture we have used is illustrated in Figure 1. A planner solves problem instances in the game domain. Solution plans are converted into scripts. These are plugged into the game using the interface that the game product offers.

In implementing the architecture, we chose the JSHOP2 planning system, a Java implementation of SHOP2 (Nau *et al.* 2003) that is publicly available at `http://www.cs.umd.edu/projects/shop/`. It can handle numerical variables, making it useful to model variables such as money amounts and hunger levels.

Our choice of using Bethesda Softworks THE ELDER SCROLLS IV: OBLIVION as a game application is supported by several reasons. It is a very popular game that accepts user input in the form of scripts using an editor program publicly available at `http://www.elderscrolls.com/downloads/updates_utilities.htm`. In this game, planning can be combined with scheduling, making it more interesting from a research perspective. Each NPC can have a set of *AI packages* and/or a script that are used to control that NPC's behavior. Each AI package describes an atomic behaviour such as eating or sleeping. Actions in our computed plans will be mapped into these atomic behaviours, providing a mechanism for plan execution in the game world. The AI packages have preconditions that can include a time range. For example, a sleep package could be scheduled to activate from 10pm to 8am. General scheduling is achieved using AI packages with associated time ranges. Traditionally, activation intervals are set manually. This is in fact manual planning (i.e., select a collection of appropriate atomic behaviours) and scheduling (i.e., select their activation times) and our goal is to automate this process. In this game, scripts have typically been used to control complicated actions such as dialogue and quest related actions. We extend the use of scripts to automatic planning, and scheduling of actions in the form of AI packages, replacing the simple manual form of scheduling already present in the game.

A human expert designs one or more HTNs specific to a given game and one or more problem instances. Under this model, the problem instance must contain details of the characters, such as what sort of tasks they can do, as well as their initial states and any goals. JSHOP2 is run for each problem instance. The script generator then automatically maps the solution plans from the standard planning language (e.g., PDDL) into the scripting language that the game accepts. Planning and script generation are performed offline. Finally, the plans represented in the scripting language are added to the game as new scripts.

This effort of designing an HTN is amortized over all scripts that are generated using that HTN. We used an HTN that divides a day into 24 discrete time units. If desired, plans that cover several days can be obtained by chaining plans that correspond to one day each. The final state of a 24-hour plan is used as the initial state of the next day's planning task.

In our planning application, a problem state contains information such as and the amount of deer, the cost of each trading item, an enumeration of all locations, a list of all stores, information about what items are traded in each store, and information about who owns each shop. In addition, a problem state contains the states of all NPCs. NPC states store the values of money, hunger and tiredness, a list of tasks that each NPC can currently perform, their current activity (e.g., working), their current location, the place where they live, a list of items in their possesion (e.g., skins), and a list of pending tasks.

The HTN used is partly illustrated in Figure 2. Ovals are

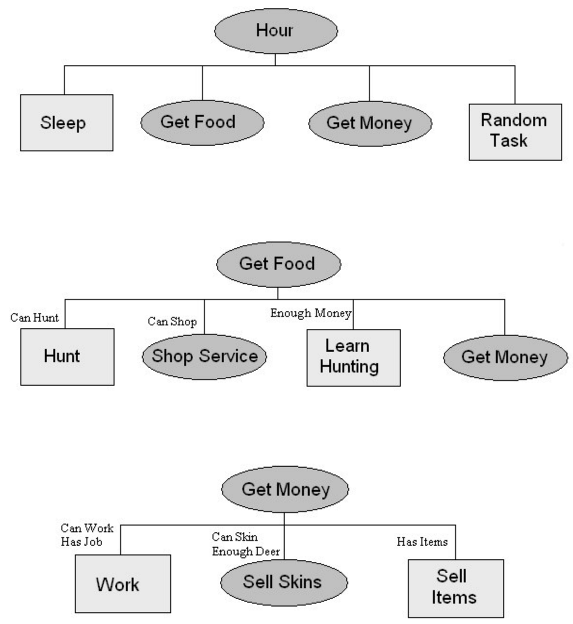

Figure 2: Part of the hierarchical task network designed for the OBLIVION game.

high-level tasks and rectangles are primitive (atomic) tasks. When refining a high-level task, the decomposition methods are tried from left to right, until a method is found whose preconditions match the current state. The picture at the top illustrates four methods (alternatives) to refine the abstract task of spending one time unit such as a game hour. If none of the first three alternatives is selected, an atomic task is picked at random, such that the schedule of the NPC at hand is not left empty for that time interval.

The picture in the middle shows decomposition alternatives for getting food. In the first two cases, the NPC can get food right away by hunting or shopping, When food cannot be obtained immediately, tasks such as learning how to hunt or making money will help to get food in the future. Paying to learn to hunt comes up when a character only knows how to shop but the store does not open. After a character has waited for long enough at the store, they temporarily remove shopping as a valid task. If they become hungry they need to get food via other means. The bottom part of Figure 2 shows the decomposition methods for getting money. The most preferred method is going to work, followed by selling skins and, as a last resort, selling other items that the NPC owns.

In classical planning, the objectives of a planning problem are defined as *reachability goals*, atoms that must hold in the final state of a plan. In HTN planning, the problem objective can be encoded either as a reachability goal or as an abstract task. All valid plans have to be a refinement of an abstract task. We have taken the latter approach. The problem objective is to compute a daily schedule such that each NPC completes tasks such as eating, sleeping, and attending their jobs.

We have developed a script generator that maps from a PDDL plan to OBLIVION's scripting language. A plan is converted into several scripts, one for each NPC involved. Scripts should not be seen as independent from each other, as NPCs typically interact in a plan. Actions in each script are totally ordered. Script code will be repeatedly run through by the game from beginning to end in a loop. To perform the correct action, the script needs to map from game times to time steps in the plan. The problem file will also need to be read, enabling the script to ensure that at the start of each day, the game state matches the initial state in the plan. Since the input plans are correct and the environment is assumed not to interfere with plan execution, no error checking needs to be encoded inside a script.

The generated scripts assume that the plan sequentially goes through the actions required of each NPC for each game hour. As the script will be run in a loop, a switch statement ensures that each set of actions is performed in the correct game hour. The script ensures that the set of actions for a game hour is performed in the correct order. For instance, first go to shop, then buy food. It also needs to ensure that each action is run to completion and that each action is only performed once. It does this by storing variables that keep track of the number of actions performed for the hour as well as a flag for the last hour that had all of the actions completed. Scripts must also encode any exchanges of goods that occur between NPCs. This is done by simple flag based message passing. An NPC will send a message to another NPC that he wants to perform an exchange. The other NPC will receive this request and perform his part of the transaction and send confirmation, at which point the first NPC can perform his part of the transaction.

NPCs perform atomic actions in the game world using the existing AI package framework. We created a set of AI packages that correspond to actions that could show up in a plan. Each atomic action in the plan is then performed by activating the appropriate AI package. The new AI packages have no time ranges assigned beforehand. Their activation time is determined according to the position of the corresponding action in the plan being executed at a given time.

Example

We show how the ideas presented before work on a small example with three characters named Bob, Sven and Fred. As Sven and Fred own one store each, they interact with Bob via tranding transactions. For the sake of simplicity, we focus our attention to one character (Bob) and to a few game hours. Part of the initial state is shown in Figure 3. The problem objective is simply stated as ((day Fred Sven Bob)). The planner will decompose this high-level task into a detailed schedule of all three characters over a period of 24 game hours.

Figure 4 shows a few steps of a solution plan. Bob is asleep, then wakes up and is hungry. He has some money so he goes to buy food from the store. The store is not open yet so Bob must wait. The storekeeper shows up soon enough that Bob has not yet given up and Bob eats. He then goes to

```
(cost food 5.0)
(cost skins -4.0)
(cost goods -50.0)
(cost learn-hunt 30.0)
(cost rent 1.0)
(shop skin-shop skins Sven)
(shop skin-shop goods Sven)
(shop food-shop food Fred)
(place Bobs-house)
(place forest)
(place skin-shop)
(place food-shop)
(deer 4.0)
(asleep Bob)
(money Bob 7.0)
(available-task Bob purchase-food
                          sedate-hunger)
(available-task Bob wander wander)
(available-task Bob skin get-money)
(available-task Bob sleep sleep)
(available-task Bob sell-goods get-money)
(hungry Bob 7.0)
(sleepy Bob 4.0)
(have-goods Bob)
(at Bob Bobs-house)
(lives Bob Bobs-house)
```

Figure 3: Part of the initial state in the running example. The first half contains "global" information such as the cost of food and the number of deer. The second half contains Bob's variables and available tasks. For simplicity, state variables that detail Fred and Sven's status are skipped.

```
(!sleep Bob 5.0)
(!increment-person Bob)
(!sleep Bob 2.0)
(!increment-person Bob)
(!wake Bob)
(!add-task Bob purchase-food)
(!move Bob food-shop)
(!wait Bob food-shop)
(!increment-person Bob)
(!shop-food-service Bob)
(!increment-person Bob)
(!add-task Bob skin)
(!move Bob forest)
(!skin Bob)
(!move Bob skin-shop)
(!shop-skin-service Bob)
```

Figure 4: Steps of a daily plan corresponding to the beginning of Bob's schedule.

make some money by going to the forest and getting some deer pelts, before returning to town to sell them. The HTN includes a framework of each NPC getting hungry and tired over time. The repeated !increment-person calls correspond to Bob getting more tired and hungry over time.

Figure 5 shows a script fragment that corresponds to the last four steps of the plan in Figure 4. A detailed explanation of the code is beyond the focus of this paper. The first line ensures that each action is scheduled at the appropriate time. The *current* variable keeps track of how many actions have been completed in the current time period. This ensures that the actions are done in the correct sequence. It would not make sense to sell the skins before acquiring them for example. The *bobVar* variable is simply in place to ensure that Bob will not repeatedly start an AI package. Once Bob moves to the forest he begins skinning. This is simply an AI package where Bob will kill a deer. The deer has scripts so that once dead, Bob will be informed. Bob has the skinning variable set to 1 so a skin appears and he will move onto the next action.

Once Bob arrives at the store, the *SellSkins* package initiates trade. The merchant will have scripts set up to figure out what is being bought or sold and perform the appropriate transaction. Bob has selling-skin set to 1 so the merchant knows to buy the skin and Bob is done. Notice that the value of *current* is incremented by the other character when the action is complete, as Bob cannot know when this action will complete after he initiates it. The last section of script simply ensures that once Bob has completed all of the actions, he will not try to do them again.

Discussion

In this section we briefly present a few insights that would need to be considered when planning is integrated in a game.

An important issue when modelling a game sub-problem as a planning task is choosing between a static, fully observable environment and an environment that can be dynamic or partially observable or both. A trade-off to consider is that algorithms for static and fully observable environments are generally faster, better studied and scale up better, whereas an environment in the second category might model a problem more realistically. In a game world, part of the problems can effectively be modeled as planning in a static environment. The previous section has shown such an example, where NPCs interact with each other but do not interact with the player character during the execution of a plan. Other scenarios would require a dynamic environment. Examples of exogenous events include the player character's actions (e.g., lock a door) and natural phenomena (e.g., it starts raining). "Fog of war", a common expression used in games, refers to incomplete information and partial observability. A classical example of incomplete information is an initially unknown map in a real time strategy game. The actual topology is gradually discovered as one or more mobile units explore new parts of the map.

Partial observability and exogenous events require a planning approach that is robust enough to handle new situations that could not be anticipated at planning time. Possible approaches from the literature that deal with incomplete information include contingency planning (Peot and Smith 1992) (i.e., compute a tree-shaped solution and choose the actual execution path according to the conditions at execution time), conformant planning (Goldman and Boddy 1996)

```
if (theTime >= 10 && theTime < 11&&
    finished != 10)
   if current == 0
      AddScriptPackage TravelDeerForest
      set current to current + 1
   endif
   if current == 1
      if bobVar != 60
         set bobVar to 60
         set skinning to 1
         set skinDeer to 0
         AddScriptPackage skin
      endif
   endif
   if current == 2
      AddScriptPackage TravelSkinShop
      set current to current + 1
   endif
   if current == 3
      if bobVar != 70
         set bobVar to 70
         set selling-skin to 1
         AddScriptPackage SellSkins
      endif
   endif
   if current == 4
      set current to 0
      set finished to 10
      set bobvar to 0
   endif
endif
```

Figure 5: Part of a script.

(i.e., compute a linear solution that covers several possible scenarios) and replanning. Contingency planning and conformant planning can be performed offline. Replanning requires an online integration of the planning system and the game engine.

In dynamic or partially observable domains, computing a plan that is guaranteed to work in all possible situations could be a tremendously hard task and could produce solutions of large size. In contrast, online planning solves a simpler problem each time (e.g., assume the environment is static), and performs a replanning step when the simplified solution doesn't match the actual game world. When performed at runtime, a resource-intensive planning process can become a performance bottleneck. Game AI usually gets limited access to the memory and CPU resources, as sophisticated graphics and sound algorithms are expensive. Guiding search with HTNs could effectively address this issue.

The development effort necessary to add planning capabilities to a game is another itssue to consider. Its importance should not be underestimated in an industry with tight deadlines, where commercial companies tend to ignore new ideas unless they can easily be added to the project being developed at the given time (Hopson 2006). As shown before, existing game components such as scripting and AI packages can greatly simplify the integration of planning. More generally, an HTN planning library could be implemented as a standard plugin component, allowing developers to take advantage of the implementation without needing extensive knowledge of the underlying model. This method of implementation has seen some success with graphics engines (Valve 2007), networking code (Gamespy 2007) and physics simulators (Havok 2007).

Let us consider how issues such as those discussed above apply to our approach. Producing scripts, whether automatically or by hand, is an offline process. This means that it is hard to account for the many possibilities that could be encountered within a non-deterministic game world. While some contingency planning is possible, it is hard to ensure that the produced scripts will work every time; for example, if a player went around killing characters vital to plan execution, it would not be able to successfully execute. This feature is inherent to offline processes. However, this does not invalidate the idea of using offline scripting and planning in games. In our test application, the plans typically remain valid unless another character (such as the player) actively tries to prevent it. In a role playing environment, there are virtual laws in place that would deter characters (including the player) from bad behavior such as killing another human-like character. This means that under normal playing circumstances, there is little that would stop plans that model the daily schedules of NPCs from executing successfully.

One main advantage to using our method comes in the form of development speed. It is possible to imitate the behaviour of a character controlled using our method by using the standard schedules and scripts. In terms of in game appearance, they would be identical, but manual planning would require a tedious effort, even for problems of moderate size. Better scripting capabilities via automated planning translate into a better gaming experience, allowing more controlled characters to populate a world and computing more complex plans, which cover longer game periods and have more NPC interactions.

Comparison with ScriptEase

To the best of our knowledge, the work on ScriptEase (e.g., (McNaughton *et al.* 2004)) is the only research on automating script generation previously reported in the games literature. ScriptEase is a tool that allows a user to generate scripting code through a graphical front end. Scripting can be performed even by users with no programming skills. The process is faster and safer, since manual coding introduces bugs that can take a long time to hunt down.

As an important difference from our work, ScriptEase makes no attempt at automating the planning side of scripting. The user is responsible to select all actions that compose a script, possibly starting from a predefined pattern. Arguably, ScriptEase could be seen as focused on the software engineering side of scripting rather than the artificial intelligence part. The authors state that AI capabilities such as learning would be a good addition to the ScriptEase

NPCs	Time (sec.)	Nodes	Plan Length
3	6.12	1108	293
4	6.49	1452	381
5	7.07	1798	469
6	7.56	2142	557
7	7.94	2486	645
15	10.83	6332	1351
20	12.47	8062	1797
40	28.57	14982	3566

Table 1: Summary of results when the number of NPCs grows and the time interval is fixed to 24 game hours.

Game Hours	Time (sec.)	Nodes	Plan Length
4	12.31	2634	700
8	15.80	4570	1416
12	17.49	6566	2134
16	20.45	9086	2857
20	24.01	11126	3591
24	25.32	13614	4308

Table 2: Summary of results when the number of NPCs is fixed to 40 and the time interval varies from 4 to 24 game hours.

tool (McNaughton *et al.* 2004). In addition, we believe that our work and SCRIPTEASE can be combined into a tool that would provide access to a planning engine through a sophisticated and intuitive graphical interface that would not require users to have deep planning knowledge. The GUI and all the machinery behind it could be used to design hierarchical task networks, and to state the initial state and the goals of specific planning tasks. Based on such input data, the planning engine can compute plans and represent them into the scripting language.

Experiments

This section contains an empirical evaluation of the implemented system. Two experiments are run using the HTN described in the previous section. The HTN consists of 19 low-level operators and 12 high-level methods. Planning for 3 NPCs over 24 hours requires a problem definition with 49 atoms in the initial state as defined in the state representation. For each new NPC, the initial state should encode its original status (e.g., amount of money) and list of available tasks, as shown in the second half of Figure 3 for character Bob.

In the first experiment, problem size varies by gradually increasing the number of NPCs from 3 to 40. Each plan covers a period of 24 game hours. In a second experiment, we fix the number of NPCs to 40 and gradually increase the covered period from 4 game hours up to 24 game hours. Tables 1 and 2 summarize the results of each experiment. We show the number of NPCs, the search effort measured both as CPU time and nodes, and the length of produced plans. All tests were run on a machine with an AMD Athlon 64 X2 4200+ processor and 1 GB RAM.

The data in Tables 1 and 2 show that planning effort grows linearly in both the number of NPCs and the number of discrete time units in the planned day.

Adding a new NPC to a problem instance requires less than 30 seconds of modifications to the problem and HTN files. The HTN needs some values adjusted to inform it that it is planning for an additional NPC. The problem file requires a specification of the initial values of the NPC as well as an enumeration of the tasks available to him. The initial values for each new NPC were generated randomly.

If we need to significantly change the behaviour of an NPC under our planning model, this simply requires changing the description of that NPC in the problem file, then running the planner and the script generator again. A description for the NPC contains the NPCs initial values and an enumeration of the tasks that it can do. This manual editing of a problem definition should take less than a couple of minutes.

Once a plan is found, the script generator will produce scripts typically in under a second per script. The script generated for each NPC varies between 80 and 500 lines, depending on the number of time steps. The plans produced are quite long but much of the plan will contain adjustment steps for state variables in the plan space that are not relevant to the game space. The game does not need to know exactly how hungry an NPC is as long as the NPC is told when to go and get food. With 40 NPCs, the total time to produce a script for each of them starting from a plan will be less than 30 seconds for 24 time slots. In comparison, the scenario to manually plan, then manually write a script of 500 lines and repeat this process 40 times is extremely tedious.

Conclusion

Planning is a powerful but potentially resource-intensive technology to add intelligent behaviour to NPCs in a game. HTNs guide search with hand-coded knowledge, reducing the planning effort considerably. In this paper we have introduced an approach to HTN planning in the domain of video games. Our implementation generates scripts automatically, replacing the traditional approach to create scripts manually. In experiments, scripts are generated at a level of complexity that would require a great human effort to compose and debug.

We are currently developing a model for online planning with HTNs and using replanning, which will result in a standard plugin library. Other ideas for future work include applying offline conformant planning and offline contingent planning to video games.

Acknowledgments

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs. Sven Koenig was on sabbatical at NICTA when this research was performed.

References

Epic Games. Unreal tournament, 2007. `http://www.unrealtournament.com`.

Gamespy. Gamespy: Multiplayer Gaming's Homepage, 2007. `http://au.gamespy.com/`.

R. P. Goldman and M. S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems AIPS-96*, pages 110–117. AAAI Press, 1996.

Havok. Titles That Use Havok Dynamics, 2007. `http://www.havok.com/content/blogcategory/29/73/`.

J. Hopson. We're Not Listening: An Open Letter to Academic Game Researchers, 2006. `http://gamasutra.com/features/20061110/hopson_01.shtml`.

O. Ilghami, D. S. Nau, H. Munoz-Avila, and D. W. Aha. CaMeL: Learning Methods for HTN Planning. In *Proceedings of the International Conference on AI Planning and Schedulling AIPS-02*, pages 131–142, 2002.

D. Isla. Handling Complexity in the Halo 2 AI. In *Proceedings of Game Developers Conference GDC-05*, 2005.

U. Kuter and D. Nau. Forward-chaining Planning in Nondeterministic Domains. In *Proceedings of the National Conference on Artificial Intelligence AAAI-04*, pages 513–518, 2004.

U. Kuter and D. Nau. Using Domain-configurable Search Control for Probabilistic Planning. In *Proceedings of the National Conference on Artificial Intelligence AAAI-05*, pages 1169–1174, 2005.

M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of the IEEE International Conference on Automated Software Engineering ASE-04*, pages 88–99, 2004.

H. Munoz-Avila and T. Fisher. Strategic Planning for Unreal Tournament Bots. In *AAAI-Workshop on Challenges on Game AI*, 2004.

D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

J. Orkin. Constraining Autonomous Character Behavior with Human Concepts. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, 2003.

J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of Game Developers Conference GDC-06*, 2006.

M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems AIPS-92*, pages 189–197, 1992.

E. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-75*, pages 206–214, 1975.

Sierra. F.E.A.R. Available Now, 2007. `http://www.whatisfear.com/us/`.

S. Smith, D. Nau, and T. Throop. Computer Bridge: A Big Win for AI Planning. *AI Magazine*, 19(2):93–105, 1998.

B. Stein. The Elder Scrolls IV: Oblivion - Radiant A.I., 2007. `http://www.xbox.com/en-US/games/t/theelderscrollsIVoblivion/20051208-radian%tai.htm`.

A. Tate. Generating Project Networks. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-77*, pages 888–893, 1977.

Valve. Source SDK Release Notes – Valve Developer Community, 2007. `http://developer.valvesoftware.com/wiki/Source_SDK_Release_Notes`.

D. Wilkins and M. desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.

D. Wilkins. *Practical Planning: Extending the Classical Planning Paradigm*. Morgan Kauffman, 1988.