Generating Ambient Behaviors in Computer Role-Playing Games

Maria Cutumisu[1], Duane Szafron[1], Jonathan Schaeffer[1], Matthew McNaughton[1], Thomas Roy[1], Curtis Onuczko[1], and Mike Carbonaro[2]

[1] Department of Computing Science, University of Alberta, Canada
{meric, duane, jonathan, mcnaught, troy,
onuczko}@cs.ualberta.ca
[2] Department of Educational Psychology, University of Alberta, Canada
{mike.carbonaro}@ualberta.ca

Abstract. Many computer games use custom scripts to control the ambient behaviors of non-player characters (NPCs). Therefore, a story writer must write fragments of computer code for the hundreds or thousands of NPCs in the game world. The challenge is to create entertaining and non-repetitive behaviors for the NPCs without investing substantial programming effort to write custom non-trivial scripts for each NPC. Current computer games have simplistic ambient behaviors for NPCs; it is rare for NPCs to interact with each other. In this paper, we describe how generative behavior patterns can be used to quickly and reliably generate ambient behavior scripts that are believable, entertaining and non-repetitive, even for the more difficult case of interacting NPCs. We demonstrate this approach using BioWare's Neverwinter Nights game.

1 Introduction

A computer role-playing game (CRPG) is an interactive story where the game player controls an avatar called a player character (PC). Quickly and reliably creating engaging game stories is essential in today's market. Game companies must create intricate and interesting storylines cost-effectively and realism that goes beyond graphics has become a major product differentiator. Using AI to create non-player characters (NPCs) that exhibit near-realistic ambient behaviors is essential, since a richer background "tapestry" makes the game more entertaining. However, this requirement must be put in context: the storyline comes first. NPCs that are not critical to the plot are often added at the end of the game development cycle, only if development resources are available. Consider the state-of-the-art for ambient behaviors in recent CRPGs. In Fable (Lionhead Studios), the NPCs wake at dawn, walk to work, run errands, go home at night, and make random comments about the disposition and appearance of the PC. However, the behaviors and comments are "canned" and repetitive and NPCs never interact with each other. The Elder Scrolls 3: Morrowind (Bethesda Softworks) has a huge immersive world. However, NPCs either wander around areas on predefined paths or stand still, performing a simple animation, never interacting with each other and ignoring the simulated day. In The Sims 2 (Electronic Arts), players control

the NPCs (Sims) by choosing their behaviors. Each Sim chooses its own behaviors using a motivational system if it is not told what to do. The ambient behaviors are impressive, but they hinge on a game model (simulation) that is integral to this game and not easily transferable to other game genres, including CRPGs. Halo 2 (Bungie) is a first person shooter with about 50 behaviors, including support for "joint behaviors" [11][20]. The Halo 2's general AI model is described, but no model for joint behaviors is given. Façade [7] has an excellent collaborative behavior model for NPCs, but there are only a few NPCs, so it is not clear if it will scale to thousands of ambient NPCs. They also comment about the amount of manual work that must be done by a writer when using their framework. Other research includes planning, PaTNets, sensor-control-action loops [1][16], and automata controlled by a universal stack-based control system [3] for both low-level and high-level animation control, but not in the domain of commercial-scale computer games. However, planning is starting to be used in commercial computer games in the context of Unreal Tournament [5][21]. Crowd control research involves low-level behaviors such as flocking and collisions [14] and has recently been extended to a higher-level behavioral engine [2]. Group behaviors provide a formal way to reason about joint plans, intentions and beliefs [10]. Our approach is dictated by the practical requirements of commercial computer games. The model we describe in this paper is robust, flexible, extendable, and scalable [6] to thousands of ambient NPCs, while requiring minimal CPU resources. Moreover, our generative pattern abstraction is essential to story designers, shielding them from manual scripting and the synchronization issues of collaborative behaviors, and allowing them to concentrate on story construction.

In most games, scripts control NPC behaviors. A game engine renders the story world objects, generates events on the objects, dispatches events to scripts and executes the scripts. Different stories can be "played" with the same game engine using story-specific objects and scripts. Programmers create game engines using programming languages such as C or C++. Writers and artists, who are not usually programmers [17], write game stories by creating objects and scripts for each story. The goal of our research is to improve the way game stories, not game engines, are created.

A writer may create thousands of game objects for each story. If a game object must interact with the PC or another game object, a script must be written. For example, BioWare Corp.'s popular Neverwinter Nights (NWN) [15] campaign story contains 54,300 game objects of which 29,510 are scripted, including 8,992 objects with custom scripts, while the others share a set of predefined scripts. The scripts consist of 141,267 lines of code in 7,857 script files. Many games have a toolset that allows a writer to create game objects and attach scripts to them. Examples are BioWare's Aurora toolset that uses NWScript and Epic Game's UnrealEd that uses UnrealScript.

The difficulties of writing manual scripts are well documented [12]. Writers want the ability to create custom scripts without relying on a set of predefined scripts or on a programmer to write custom scripts. However, story creation should be more like writing than programming, so a writer should not have to write scripts either. A tool that facilitates game story writing, one of the most critical components of game creation, should: 1) be usable by non-programmers, 2) support a rich set of non-repetitive interactions, 3) support rapid prototyping, and 4) eliminate most common types of errors. ScriptEase [19] is a publicly available tool for creating game stories using a high-level menu-driven "programming" model. ScriptEase solves the non-

programmer problem by letting the writer create scenes at the level of "patterns" [9][13]. A writer begins by using BioWare's NWN Aurora toolset to create the physical layout of a story, without attaching any scripts to objects. The writer then selects appropriate behavior patterns that generate scripting code for NPCs in the story. For example, in a tavern scene, behavior patterns for customers, servers and the owner would be used to generate all the scripting code to make the tavern come alive.

We showed that ScriptEase is usable by non-programmers, by integrating it into a Grade 10 English curriculum [4]. The version of ScriptEase that was used had a rich set of patterns for supporting interactions between the PC and inanimate objects such as doors, props and triggers. It also had limited support for plot and dialogue patterns (the subject of on-going work). In this paper, we describe how we have extended the generative pattern approach of ScriptEase to support the *ambient behaviors* of NPCs.

NPC interactions require concurrency control to ensure that neither deadlock nor indefinite postponement can occur, and to ensure that interactions are realistic. We constructed an NPC interaction concurrency model and built generative patterns for it. We used these patterns to generate all of the scripting code for a tavern scene to illustrate how easy it is to use behavior patterns to create complex NPC interactions. The ambient background includes customers, servers and an owner going about their business but, most importantly, interacting with each other in a natural way. In this paper, we describe our novel approach to NPC ambient behaviors. It is the first time patterns have been used to generate behavior scripts for computer games. The research makes three key contributions: 1) rich backgrounds populated with interacting NPCs with realistic ambient behaviors are easy to create with the right model, 2) pattern-based programming is a powerful tool and 3) our model and patterns can be used to generate code for a real game (NWN). We also show that the patterns used for creating the tavern scene can be reused for other types of NPC interactions. Finally, we show how ambient behavior patterns are used to easily and quickly regenerate and improve all of the behaviors of all ambient NPCs in the NWN Prelude.

2   Defining, Using, and Evaluating Ambient Behavior Patterns

A standard CRPG tavern scene can be used to demonstrate ambient behavior patterns. We focus on three ambient behavior patterns from this scene: owner, server, and customer. More complex behaviors could be defined, but these three behaviors already generate more complex behavior interactions than most NPCs display in most CRPGs. In this section, we describe the basic behaviors generated for these patterns, how a story writer would use the patterns and how the patterns were evaluated.

Each pattern is defined by a set of behaviors and two control models that select the most appropriate behavior at any given time. In general, a behavior can be used *proactively* (P) in a spontaneous manner or *reactively* (R) in response to another behavior. Table 1 lists all of the behaviors used in the tavern. Some behaviors are used independently by a single NPC. For example, posing and returning to the original scene location are *independent behaviors*. This paper addresses only high-level behaviors, since the NWN game engine solves low-level problems. For example, if the original location is occupied by another creature when an NPC tries to return, the

game engine moves the NPC as close as possible. Subsequent return behaviors may allow the NPC to return to its original location. Behaviors that involve more than one NPC are *collaborative* (joint) *behaviors*. For example, an offer involves two NPCs, one to make the offer and one to accept/reject it. The first column of Table 1 indicates whether a proactive behavior is independent or collaborative. Note that interactions with the PC are not considered ambient behaviors and they are not discussed in this paper. The most novel and challenging ambient behaviors are the ones that use behaviors collaboratively (interacting NPCs). The second column lists the proactive behaviors. The letters in parentheses indicate which kind of NPC can initiate the proactive behavior. For a collaborative behavior, the kind of collaborator is given as part of the behavior name, e.g., the "approach random C" behavior can be initiated by a server or customer (S, C) and the collaborator is a random customer (C).

Table 1. Behaviors in the Server (S), Customer (C), and Owner (O) Patterns

Behavior Type	Proactive Behavior	Reactive Chains	
Independent	pose (S, C, O)	pose, done	
	return (C, O)	return, done	
	approach bar (S, C)	approach, done	
	fetch (O)	fetch, done	
Collaborative	approach random C (S, C)	approach, done	
	talk to nearest C (C)	speak, speak, converse*	
	converse with nearest C (C)	(speak, speak)+ done	
	ask-fetch nearest S (C, O)	speak, fetch, receive, speak, done	
	ask-give O (C)	speak, give, receive, speak, done	
	offer-give to nearest C (O)	speak, decide, ask-give*;	*(accept)*
		speak, decide, speak, done	*(reject)*
	offer-fetch to nearest C (S)	speak, decide, ask-fetch*;	*(accept)*
		speak, decide, speak, done	*(reject)*

The third column of Table 1 shows the reactive chains for each proactive behavior. For example, the *ask-fetch* proactive behavior generates a reactive chain where the initiator *speaks* (choosing an appropriate one-liner randomly from a conversation file), the collaborator *fetches* (goes to the supply room while speaking), the initiator *receives* something, the collaborator *speaks* and the *done* behavior terminates the chain. Each reactive chain ends in a *done* behavior, unless another chain is reused (denoted by an asterisk such as *converse** in the *talk* behavior). Each behavior consists of several actions. For example, a *speak* behavior consists of facing a partner, pausing, performing a speech animation and uttering the text. A *()+* indicates that the parenthesized behaviors are repeated one or more (random) times. For example, the *converse* proactive behavior starts a reactive chain with one or more *speak* behaviors, alternating between two characters. The *talk* proactive behavior starts a reactive chain with a *speak* behavior (a greeting) for each interlocutor, followed by a *converse* behavior. The *offer-give* (owner offers a drink) and *offer-fetch* (server offers to fetch a drink) proactive behaviors each have two different reactive chains (shown in Table 1) depending on whether the collaborator *decides* to accept or reject the offer.

The writer uses the Aurora toolset to construct the tavern area, populate it with customers, servers and an owner, and saves the area in a module. The writer then opens

the module in ScriptEase and performs three kinds of actions. First, create some instances of the server, customer and owner patterns by selecting the patterns from a menu and then binding each instance to an appropriate NPC. Second, bind the options of each pattern instance to game objects and/or values. Fig. 2 shows how to set options (gray tabs in the left pane) for the server NPC. The center pane shows how the `Actor` option is bound to a `Server` NPC, created in the Aurora toolset. The right pane shows how to (optionally) change the relative default proactive behavior chances described in Section 3. The spin chances do not need to add to 100 – they are automatically normalized. Third, select the "Save and Compile" menu command to generate 3,107 lines of NWScript code (for the entire tavern scene) that could be edited in the Aurora toolset if desired. The simplicity of the process hides the fact that a large amount of scripting code is generated to model complex interactive behaviors.
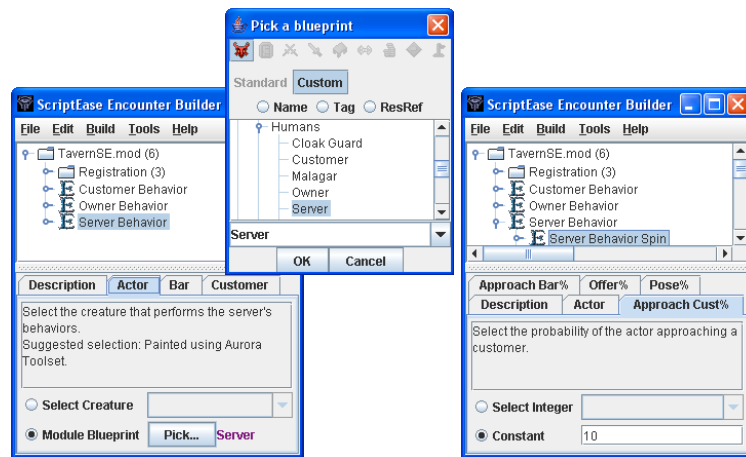

Fig. 1. Using ScriptEase ambient behavior patterns for a tavern scene

The behavior patterns are easy to use - creating and testing a tavern scene in NWN required less than half an hour. The generated code is efficient, producing ambient behaviors that are crisp and responsive, with no perceptible effect on response time for PC movement and actions. The NPCs interacted with each other flawlessly with natural movements. A scene with ten customers, two servers and an owner was left to play for hours without any deadlock, degradation in performance, repetition or indefinite postponement of behaviors for any actor. Since the effectiveness and performance of ambient behaviors is best evaluated visually, we illustrate our approach using a series of movies captured from actual game-play [19]. These patterns were designed for a tavern scene. However, they are general enough to generate scripts for other scenes. For example, in a house scene, the customer pattern can be used for the inhabitants, the server pattern for a butler, and the owner pattern for a cook. The butler interacts with the inhabitants, fetching for them by going to the kitchen. The inhabitants talk amongst themselves and the cook occasionally fetches supplies. Our approach handles group (crowd) behaviors in a natural way. The customers constitute an example of a crowd – a group of characters with the same behavior, but each selecting different behaviors based on local context.

To determine the range of CRPG ambient behaviors that can be accommodated by patterns, we conducted a case study for the Prelude of the NWN official campaign, directed at both independent and collaborative behaviors. The original code used ad-hoc scripts to simulate collaborative behaviors. We removed all of the manually scripted NPC behaviors and replaced them with behaviors generated from patterns. Six new ambient behavior patterns were identified: *Poser*, *Bystander*, *Speaker*, *Duet*, *Striker*, and *Expert*. These patterns were sufficient to generate all of the NPC ambient behavior scripts. Further evidence of the generality of ambient behavior patterns will require a case study that replaces behaviors in other game genres as well. There is no reason why a soccer or hockey goaltender could not be provided with entertaining ambient behaviors to exhibit when the ball (puck) is in the other end of play, such as standing on one leg, stretching, leaning against a goal post, or trying to quiet the crowd with a gesture. For example, one of the criticisms for EA FIFA 04 was directed to the goalie's behavior [18] and will be addressed in the announced EA FIFA 06 [8].

3   Creating New Ambient Behavior Patterns

To create new behavior patterns or adapt existing behavior patterns, one must look one level below the pattern layer at how the patterns are constructed from *basic behaviors*. A pattern designer can compose reusable basic behaviors to create a new behavior pattern or add basic behaviors to existing patterns, without writing any scripts. It is easy to mix/combine behaviors. There are two more levels below the pattern construction layer – the concurrency control and the script layers.

Each behavior pattern includes a proactive model and a reactive model. The *proactive* model selects a *proactive behavior* based on probabilities. This simplest proactive model uses static probabilities assigned by the writer. For example, the server pattern consists of the proactive behaviors *approach* a random customer, *approach* the bar, *offer-fetch* a drink to the nearest customer and *pose*. In this case, a static probability distribution function [.10, .05, .03, .82] could be used to select one of these behaviors for each proactive event. The left pane of Fig. 2 shows the proactive model for the server. The *reactive* model specifies a *reactive chain* for each proactive behavior. For example, the right pane of Fig. 2 shows the reactive chain for the server's *offer-fetch* proactive behavior listed in Table 1. Each reactive behavior fires an event that triggers the next reactive behavior until a *done* behavior signals the end of the reactive chain. The circle identifies the actor that performs the behavior (S, server; C, customer). Other options, such as what is spoken, have been removed from the diagram for clarity. Each of the other three proactive behaviors for the server (*approach bar*, *approach customer,* and *pose*) has a reactive chain that consists of a single behavior followed by a *done* behavior, as listed in Table 1.

A behavior can use selection to choose between multiple possible following behaviors. For example, the *decide* behavior can fire either one of two *speak* events based on the customer's drink wishes. A loop can be formed when a behavior later in the chain fires an event earlier in the chain. Loop termination can result from using selection to exit the loop. In general, the reactive model could be a cyclic graph, providing complete expressive power. For ambient behaviors, loops do not appear to be neces-

sary – reactive chains (with decision points) seem to be sufficient. For non-ambient behaviors, these loops may be necessary. Each proactive behavior that has reactive components serves as an entry point into the reactive model. The simplicity of the reactive model hides a necessarily complex concurrency model underneath (described in Section 4). The basic behaviors we created for the tavern scene (*speak*, *decide*, *receive* etc.) provide sufficient reusable components to create other ambient behavior patterns. However, it is easy to create new reusable basic behaviors as well. A new basic behavior is a series of simple ScriptEase actions, such as move to a location/object or face a direction. If no new basic behaviors are required, a new behavior pattern can be constructed in about an hour. Each new basic behavior could also take about an hour to complete. Once made, basic behaviors can be reused in many behavior patterns and behavior patterns can be reused in many stories. ScriptEase contains a pattern builder that allows a pattern designer to create new encounter patterns. We have added support to it for building basic behaviors and ambient behavior patterns.
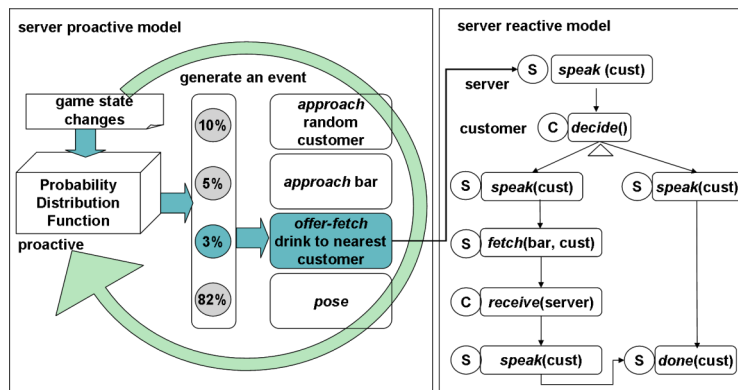

Fig. 2. The proactive model and a reactive chain for the Server pattern's *offer-fetch* behavior

Our proactive model also supports complex decisions, based on motivation or context so that it can be used for NPCs that are more important to the story. In each case, the probabilities for each proactive behavior are dynamic, based on either the current motivations (state of the NPC) or the context (state of the world). However, in this paper we focus on a static probabilistic proactive model – most NPC "extras" do not need motivational models to control their ambient behaviors.

4   The Concurrency Control Model

Concurrency models have been studied extensively for general-purpose computing. A description of the difficulties in building a concurrency model for interacting NPCs is beyond the scope of this paper. However, we raise a few points to indicate the difficulty of this problem. First, synchronization between actors is essential so that an actor completes all of the actions for an event before the next event is fired. For example, the server should not fetch a drink before the customer has decided whether to

order a drink or not. Second, deadlock must be avoided so a pair of actors does not wait forever for each other to perform a behavior in a reactive chain. Third, indefinite postponement must be avoided or some behaviors will not be performed.

Our concurrency control mechanism is invisible to the story writer and is only partially visible to the pattern designer. It has *proactive* and *reactive* components that use proactive and reactive events respectively (user-defined events are used in NWN). The *proactive* model has a proactive controller. When the PC enters an area, the controller triggers a *register proactive* event on each NPC within a range of the PC. There is no need to control ambient behaviors in areas not visible to the user, since doing so slows down game response. In games such as Fable, NPCs uphold their daily routine whether the user can see them or not. Computational shortcuts are needed to minimize the overhead. On each NPC, the registering proactive event triggers a spin behavior that, in turn, fires a single proactive event (for instance, *offer-fetch*) as a result of a probabilistic choice among all the proactive behaviors that the actor could initiate. The selected proactive event (*offer-fetch*) fires a single *reactive* event that corresponds to the first behavior in the reactive chain (*speak*). To follow the chain properly, each behavior event (proactive or reactive) has one additional string parameter called the *context*. As its last action, the basic behavior for each event (except *decide* and *done*) fires a reactive event with this context as a parameter. The pattern designer creates a reactive chain by providing suitable context values in the correct order for the desired chain. For example, to construct the *ask-fetch* chain from Table 1, the designer provides the context parameters: "speak", "fetch", "receive", "speak", "done". The *decide* behavior returns its context parameter with either "-yes" or "-no" appended so the reactive event can select the next appropriate event.

This reactive control model ensures synchronization in a single chain by preventing an actor from starting a behavior before the previous behavior is done. However, it does not prevent synchronization problems due to multiple chains. For example, suppose the server begins the reactive chain for the *offer-fetch* proactive behavior shown in Fig. 1 by *speaking* a drink offer, and suppose the owner starts a proactive *ask-fetch* behavior to send the server to the supply room. The server will receive events from both its own reactive *offer-fetch* chain and the owner's reactive *ask-fetch* chain in an interleaved manner that violates synchronization.

To ensure synchronization, we introduced an *eye-contact protocol* that ensures both actors agree to participate in a collaborative reactive chain before the chain is started. actor_1 suspends all proactive events and tries to make eye-contact with actor_2. If actor_2 is involved in a reactive chain, actor_2 denies eye-contact by restarting actor_1's proactive events. If actor_2 is not involved in a reactive chain, actor_2 sends a reactive event to actor_1 to start the appropriate reactive chain. This protocol cannot be implemented with events alone, so we use state variables of the actors.

We use another mechanism to eliminate deadlock and indefinite postponement. Either of these situations can arise in the following way. First, an eye-contact is established with an actor, so that the proactive controller does not generate another proactive event. Second, at the conclusion of the reactive chain started by the eye-contact, the actor is not re-registered to generate a new proactive event. Not only will this actor wait forever, but the other actor in the collaborative reactive chain can wait forever as well. One way for this situation to occur is for a script to clear all of the actions in an actor's action queue, including an expected action to fire an event in the

reactive chain. In this case, the reactive chain is broken and the proactive controller will never generate another proactive event for the NPC. For example an NPC's action queue is cleared if the user clicks on an NPC to start a conversation between the PC and the NPC. Our solution uses a heartbeat event to increment a counter for every NPC and to check whether the counter has reached a specific value. The game engine fires a heartbeat event every 6 seconds. If the counter reaches a threshold value, that NPC's ambient proactive controller is restarted. The counter is reset to zero every time an event is performed by the NPC, so as long as the NPC is performing events (not deadlocked) no restart will occur. Neither the story writer nor the pattern designer need be aware of these transparent concurrency control mechanisms.

We have recently added a perceptive model to our system. The perceptive model allows NPCs to be aware of the PC's presence and act accordingly. When an NPC who is performing its proactive/reactive behaviors perceives the PC, the NPC's action queue is cleared, proactive behavior generation is suspended, and the NPC performs an appropriate perceptive behavior. After the perceptive behavior is completed, proactive behavior generation resumes. This model allows NPC behaviors to be interrupted and it also supports NPC-PC interactions in addition to NPC-NPC collaborations. The success of this exercise has shown the robustness and flexibility of our proactive and reactive models, and of the underlying concurrency control mechanism.

5 Conclusion

We described a model for representing NPC ambient behaviors using generative patterns that solves the difficult problem of interacting NPCs. We implemented this model in the NWN game using ScriptEase generative patterns. We are building a common library of rich ambient behavior patterns for use and reuse across CRPGs. Our next goals are to develop patterns that support NPCs that are more central to the plot of the game and NPCs that act as henchmen for the PC. Each of these goals involves escalating challenges, but we have constructed our ambient behavior model with these challenges in mind. For example, the model supports the non-deterministic selection of behavior actions based on game state. For ambient behaviors this approach can be used with a static probability function to eliminate repetitive behaviors that are boring to the player. For non-ambient behaviors these probabilities can be dynamic and motivation-based for more challenging opponents and allies. We have constructed a synchronization model that is scalable to the more complex interactions that can take place between major NPCs and between these NPCs and the PC. We demonstrated our approach using a real commercial application, BioWare Corp.'s Neverwinter Nights game. However, our model could have a broader application domain that includes other kinds of computer games, synthetic performance, autonomous agents in virtual worlds, and animation of interactive objects.

References

1. Badler, N., Webber, B., Becket, W., Geib, C., Moore, M., Pelachaud, C., Reich, B., and Stone, M.: Planning and Parallel Transition Networks: Animation's New Frontiers. In Computer Graphics and Applications: Pacific Graphics '95 (1995) 101-117
2. Caicedo, A., Thalmann, D.: Virtual Humanoids: Let Them Be Autonomous without Losing Control. In the 4th Conference on Computer Graphics and Artificial Intelligence (2000)
3. Capin, T.K., Pandzic, I.S., Noser, H., Thalmann, N. M., and Thalmann, D.: Virtual Human Representation and Communication in VLNET. IEEE Computer Graphics and Applications. 17(2) (1997) 42-53
4. Carbonaro, M., Cutumisu, M., McNaughton, M., Onuczko, C., Roy, T., Schaeffer, J., Szafron, D., Gillis, S., Kratchmer, S.: Interactive Story Writing in the Classroom: Using Computer Games. In Proceedings of the International Digital Games Research Conference (DiGRA 2005). Vancouver, Canada (2005) 323-338
5. Cavazza, M., Charles, F. and Mead, S.J.: Interacting with Virtual Characters in Interactive Storytelling. In ACM Joint Conference on Autonomous Agents and Multi-Agent Systems. Bologna, Italy (2002) 318-325
6. Charles, F. and Cavazza, M.: Exploring the Scalability of Character-based Storytelling. In ACM Joint Conference on Autonomous Agents and Multi-Agent Systems (2004) 872-879
7. Mateas, M. and Stern, A.: Façade: An Experiment in Building a Fully-Realized Interactive Drama. Game Developers Conference (GDC 2003), Game Design Track (2003)
8. GameSpot EA FIFA Soccer 2006: http://www.gamespot.com/xbox360/sports/fifa2006/preview_6125667.html
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley (1994)
10. Grosz, B. and Kraus, S.: Collaborative Plans for Complex Group Actions. Artificial Intelligence. 86 (1996) 269 -358
11. Isla, D.: Handling Complexity in the Halo 2 AI. Game Developers Conference (GDC 2005)
12. McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J., Parker, D.: ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In Proceedings of the 19th IEEE Conference on Automated Software Engineering (ASE 2004) 88-99
13. McNaughton, M., Redford, J., Schaeffer, J. and Szafron, D.: Pattern-based AI Scripting using ScriptEase. In Proceedings of the 16th Canadian Conference on Artificial Intelligence (AI 2003). Halifax, Canada (2003) 35-49
14. Musse, S. R., Babski, C., Capin, T. K., and Thalmann, D.: Crowd Modelling in Collaborative Virtual Environments. In Proceedings of ACM Symposium on VRST (1998) 115-123
15. Neverwinter Nights: http://nwn.bioware.com
16. Perlin, K. and Goldberg, A.: Improv: A System for Scripting Interactive Actors in Virtual Worlds. In Proceedings of SIGGRAPH 96. New York. 29(3) (1996) 205-216
17. Poiker, F.: Creating Scripting Languages for Non-programmers. AI Game Programming Wisdom. Charles River Media (2002) 520-529
18. Review Amazon, EA FIFA Soccer 2004: http://www.amazon.com/exec/obidos/tg/detail/-/B00009V3KK/104-2888679-3521549?v=glance
19. ScriptEase (2005): http://www.cs.ualberta.ca/~script/scriptease.html
20. Valdes, R.: In the Mind of the Enemy: The Artificial Intelligence of Halo 2 (2004): http://stuffo.howstuffworks.com/halo2-ai.htm
21. Young, R. M.: An Overview of the Mimesis Architecture: Integrating Intelligent Narrative Control into an Existing Game Environment. In AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment, USA (2001)