An Architecture for Game Behavior AI: Behavior Multi-Queues

Maria Cutumisu, Duane Szafron

Department of Computing Science, University of Alberta
Edmonton, Canada
{meric, duane}@cs.ualberta.ca

Abstract

We describe an AI behavior architecture that supports responsive collaborative interruptible and resumable behaviors using behavior queues. This architecture wraps sets of behaviors into roles, which provide a simple efficient mechanism for encapsulating behaviors into components that can change dynamically, based on environmental criteria. To illustrate the viability of this architecture in a commercial setting, we implemented this architecture in BioWare Corp.'s *Neverwinter Nights* game. To demonstrate the usability of this architecture by game designers, we implemented a simple interface in ScriptEase, so that the architecture can be utilized with no coding skills.

Introduction

Games need an AI architecture that supports the straightforward creation and reuse of complex character behaviors. Since most games rely on custom scripts for non-player characters (NPCs), only plot-essential NPCs are scripted, resulting in repetitive behaviors for other NPCs. Since collaboration is hard to script, it is rare for NPCs to interact with each other. Behaviors are hand-scripted, non-resumable, and dependent on the game engine. With each layer of realism, complexity increases and the tools fail.

We identify five features that behaviors should exhibit. A behavior should be *responsive* (react quickly to the environment), *interruptible* (suspendible by other behaviors or events), *resumable* (continue from the point of interruption), *collaborative* (initiate and respond to joint behavior requests), and *generative* (easy to create by non-programmers). We show how our AI architecture supports all five features simultaneously. This paper combines an architectural solution to traditional game AI challenges: interruptible, resumable, responsive, and collaborative behaviors with high-level behavior constructs accessible to game designers with no programming knowledge to significantly reduce the content bottleneck for behavior creation.

Related Work

Other AI architectures fail to support at least one of the five features. *Finite state machines* (FSMs; Houlette and Fu 2003), *hierarchical state machines* (HSMs; Harel 1987) and *behavior scripts* are the most popular game techniques for telling an NPC how to behave. Two major drawbacks of FSMs are their inability to store state that can be returned to later and their failure to capture high-level patterns that reoccur frequently, such as sequences and conditionals. Therefore, FSM techniques do not scale well and their representation makes behaviors difficult to outline. It is particularly difficult to code collaboration using FSMs. There are a number of extensions to FSMs that support more complex behaviors, including *Stack-based FSMs* (for interruption and return), *Fuzzy State Machines*, *Hierarchical FSMs* (super-states can be used to share and reuse transitions), and *Probabilistic FSMs*. Neither FSMs nor their extension HSMs are resumable. Games that use FSMs include the *Quake* series and *Warcraft III, Pac-Man* (the ghost), *Assasin's Creed* (in which the guard behaviors are predictable). HSMs are used in *Destroy All Humans 2*. The disadvantage of this approach is that the hierarchy and structure of the graph do not reflect priorities very explicitly. Without support for priorities, the responsive feature is hard to achieve.

Simbionic (Fu, Houlette and Ludwig 2007) is a visual AI authoring tool that represents behaviors using FSMs with four additional features: expression evaluation (conditions are used to evaluate transitions), hierarchical or stack-based execution (one state can refer to another FSM and they can enable behavior priorities), behavior interrupt handling, and polymorphism. *Simbionic* enables users to define conditions and actions (i.e., FSM's states and transitions, respectively), as building blocks for behaviors (i.e., constructed FSMs by the user through linking actions with conditions). *Simbionic* builds behavior graphs from behavior components. The behavior graphs can be indexed via a descriptor hierarchy to enable polymorphic selection of behaviors. However, as is the case with most purely visual tools, complexity of behaviors can render these tools impractical. Although this tool provides support for four of

the five features (responsive, interruptible, resumable, and generative), it is not clear that this system supports collaboration, which is essential for providing a basic level of character believability. Support for collaboration has a large impact on the mechanisms used to implement the four features and supporting it usually involves extensive changes to the core AI architecture and has a significant impact on the quality and utility of code-generating author tools.

Behavior trees (BTs; Isla 2005) and *planners* are starting to appear in some games. Behavior trees (or hybrid HSMs) make individual states modular to support reuse. They generalize HSMs, since they allow the programmer to add code in arbitrary nodes at any level in the HSM tree. The tree reveals the overall priorities more intuitively than FSMs, and the hierarchy of behaviors is more obvious: the behavior sequences are mid-level and the behavior selector is at the top level. The *Halo* series uses BTs to implement joint behaviors, but *Halo* BTs are not resumable. Joint behaviors are not explained in the *Halo* series in detail. *Left 4 Dead* uses HSM/BTs, but the allies in this game do not follow the player character (PC) even when the player shoots enemies, so these behaviors are not responsive.

An *AI Planner* finds an action sequence that achieves a goal. However, planning does not remember past actions. The first action game to use *Hierarchical task network planning* (HTN; Sacerdoti 1975) is *Killzone 2* (Champandard, Verweij and Straatman 2009). *Real-time* (RT) *planning* is used in *F.E.A.R.* (Orkin 2006) and *No One Lives Forever 2*, but resumable behaviors are not supported. Instead, behaviors are re-planned and performed from the beginning. Planners such as *PaTNets* (Badler et al. 1995) are used in related domains, such as robotics and sensor-control applications. However, these techniques have not been successfully applied in the domain of commercial-scale computer games. *Soar* and *ACT-R* support planning and have been used to construct *Quake* bots, but have not been adopted for commercial games.

The Sims 2 (Electronic Arts 2005) have impressive ambient behaviors, but the game model is not easily transferable to other genres. Game objects broadcast interaction information to the NPCs. Each Sim responds autonomously to these broadcasts, although often player intervention is necessary to keep them on the right track.

Neverwinter Nights (*NWN*) and *Morrowind* use scripting to implement behaviors, but do not support true collaboration. Collaborative behaviors are simulated in *NWN* by placing a script on one of the NPCs that controls the actions of both. *Oblivion* uses goal-directed AI that only supports very simple collaborations. However, along with *NWN* and *Morrowind*, *Oblivion* does not support resumable behaviors. Instead, it restarts behaviors if they are interrupted. None of these systems are generative, so they are impractical for non-programmers.

A visual scripting tool that addresses the issue of game designers who are not programmers was used for an NBA game (Schwab 2008). The tool has an underlying data-driven AI system based on abstract state machines. The actors can have various roles that are activated by specific conditions. These behaviors have priority-based interrupts. The reusable roles can be assigned to several agents at once. Behavior activation can start immediately or behaviors can be enqueued for later execution. However, since this tool is not publicly available, a more in-depth comparison with our work could not be performed. It is also not clear how integral the NBA game theme is to the architecture. For example, the author mentions that parts of the system, such as perception (e.g., game state variables like *the distance to the ball*) and low level animation helpers are still "code-based" rather than data-driven. Therefore, we do not know whether the architecture is general enough to be used in non-sports domains. The author mentions that the learning curve required by the *Situation editor* is one week for programmers and longer for non-programmers, so this system may or may not be generative enough for general use.

In this paper, we describe how a *behavior multi-queue* architecture can be used to support our four behavior architectural requirements: responsive, interruptible, resumable, and collaborative, and how the architecture connects to *ScriptEase* (www.cs.ualberta.ca/~script) to support generative behaviors. Our architecture can be used to augment FSMs, HSMs, scripts, BTs, or HTNs.

Consider a tavern *Patron* NPC whose behaviors include a *Converse talk* behavior to talk to another NPC about a topic (e.g., "weather"). The *Converse talk* behavior has three consecutive tasks: *Move* to the NPC, *Exclaim*, and *Listen*. In addition, the *Patron* has another behavior to *Order* a drink from a tavern *Server*. The *Order* behavior has four phases: *Converse talk* with the *Server* to place the order, *Wait* until the *Server* returns, *Exchange* money for the drink, and *Converse talk* with the *Server* to express thanks. If the *Server* comes near the *Patron* and the *Patron* has not had a drink for some time, then the *Order* behavior should interupt a *Converse talk* behavior with another NPC. Figure 1 and Figure 2 show HSM and BT representations, respectively.

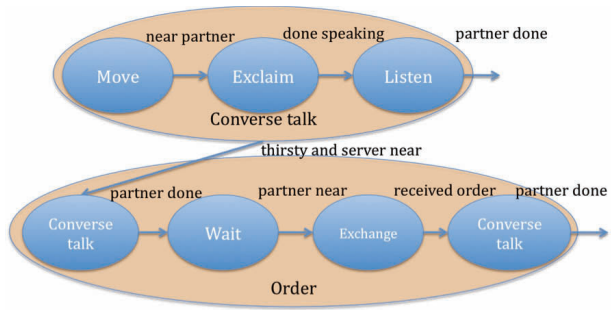

Figure 1 A partial HSM for a tavern *Patron* NPC

We use a BT notation (Champandard 2007) where a horizontal arrow above the sub-behaviors indicates that they are performed in sequence from left to right. There is no need for transitions since a sub-behavior starts when the previous one ends. The numbers under the *behavior selector* represent priorities, so that the *Order* behavior has

precedence over the *Converse talk* behavior. However, independently of the notation, there are two interpretations for these priorities. First, as soon as the conditions for the behavior with the higher priority are satisfied, this behavior interrupts any behavior with lower priority. Second, the NPC waits until a previous behavior has ended to pick a new behavior. It is the first interpretation we want for this scenario, but sometimes the second is desired.

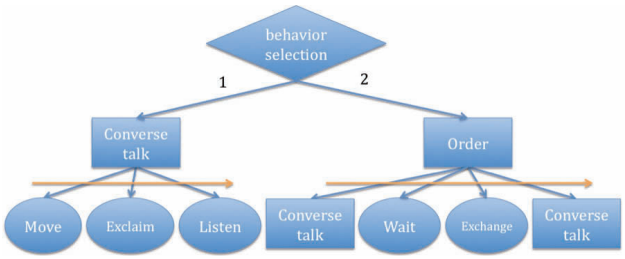

Figure 2 A partial behavior tree for a tavern *Patron* NPC

In general, many interpretations of behavior selection are possible, including five common ones (Isla 2005):

prioritized-list: the first one that can run is performed, but higher-priority siblings can always interrupt the winner on subsequent ticks.

sequential: run each in order, skipping those that are not currently relevant. When we reach the end of the list, the parent behavior is finished.

sequential-looping: same as above, but when we reach the end of the list, we start again.

probabilistic: a random choice is made from among the relevant children.

one-off: pick in a random or prioritized way, but never repeat the same choice

BTs are behavior DAGs in which the non-leaf nodes make decisions about what children to run and the leaf behaviors run the composing actions. Decision-making priorities can be computed by the parent node or by the children. The second option scales better and it is used for the core components of the combat cycle in *Halo 2*. However, there is a scalability issue, since tweaking floating point priorities by hand is difficult when there are many children, so it is usually hard to achieve the exact behavior desired by the game designer.

Both HSMs and BTs can support interruption, but behavior resumption must be added in an ad-hoc way. For example, if a thirsty *Patron* is performing a *Converse talk* behavior with another NPC about the topic "weather" and a *Server* comes close, then both the HSM of Figure 1 and the BT of Figure 2 can service the interrupt and start the *Order* behavior. However, neither provides a mechanism to resume the *Converse talk* behavior, let alone resuming it at the correct task. In the HSM situation, it could be even more difficult to develop an ad-hoc solution, since the *Converse talk* behavior appears as a sub-behavior in the *Order* behavior. Therefore, any internal state in the interrupted *Converse talk* behavior, such as the partner, will be erased (the states are not re-enterable).

Ad-hoc solutions remember the previous context in the HSM or BT, but the interrupted behavior may itself be interrupted. For example, what if the designer wanted to allow the *Server* to be interrupted by a "higher priority" behavior while performing the *Order* behavior? For example, if the PC is perceived, perhaps the *Patron* should approach the PC and initiate a conversation to start a quest. In general, we need an architectural solution to the problem of interrupting and resuming behaviors that uses a stack.

In *GTA Chinatown Wars*, Rockstar (Laming 2009) introduced a three-stack model to solve the problem of resuming interruptible behaviors. For example, if a character is going to a car and spots an enemy, the *going* behavior whose actions are on one stack is interrupted, all behaviors on this stack are popped off (except the top-level behavior), the character deals with the enemy using a second stack and, when this is done, the *going* behavior is re-started from the beginning on the first stack (not resumed from the point of interruption).

NPC collaboration is hard to express using HSMs or BTs. If each collaborator has a separate behavior, then extra code must be used to synchronize the behaviors. If a single joint behavior is used to control the collaboration, then it is difficult to integrate the joint behavior with the single behaviors of the collaborating NPCs. Collaboration has been done successfully using BTs in *Halo 3* using extra state in a blackboard (Dyckhoff 2007), but the behavior tree architecture itself provides no support. More rigorous support for collaborative behaviors has been proposed using HTNs (Gorniak 2007), where a single network structure is used to control the collaborative plan, but this work has not appeared yet in a shipped game and it has problems dealing with failed plans. If one or more collaborators fail, the other collaborators should abandon the collaboration gracefully. For example, if the *Patron* who ordered a drink leaves the tavern, the tavern *Server* should give up trying to deliver the drink.

Architectural support for interruptible-resumable behaviors and collaborations should take advantage of a potential for feature interaction. When a collaborator is waiting for its partner during a collaboration, there should be the opportunity for the waiting collaborator to suspend its own waiting behavior, perform another behavior, and resume the original behavior when the collaborator responds. For example, if a tavern *Patron* is waiting for a *Server* to fetch a drink order, then the *Patron* should be able to suspend the *Wait* task by engaging in a *Converse talk* collaborative behavior with another NPC. When the *Server* returns, the *Patron* should interrupt the *Converse talk* behavior to exchange money for the drink delivered by the *Server* and then, since this behavior is complete, resume the *Converse talk* collaborative behavior.

Façade (Mateas and Stern 2003) has a collaborative interruptible and resumable behavior model. The NPCs use collections of reactive behaviors (*beats*) that can be interrupted by the PC. Parallel beats are used for

collaboration. However, the *Façade* designers comment on the amount of manual work required from an author to use their framework, since the ABL scripting language is challenging even for experienced coders. Since *Façade* only has two NPCs in a small environment, their behavior techniques will not easily scale to hundreds or thousands of NPCs. We need a mechanism to provide behaviors of this quality to many more NPCs with minimal authoring work.

In summary, we seek architectural support for:

prioritized interruptible-resumable behaviors: a behavior may interrupt another if its priority is higher and, after this second behavior is complete, the first behavior will resume as close to the interuption as possible.

multi-level interruptions and resumptions: a chain of interruptible-resumable behaviors should be supported.

collaborative behaviors: it should be easy to specify that two or more collaborators should behave in a synchronized manner and to specify which sub-behaviors should be synchronized.

behaviors while waiting on collaborative behaviors: if a collaborator is waiting, it should be able to engage in another behavior while waiting.

graceful failures of collaborative behaviors: if one or more collaborators fail, the others should be able to gracefully abandon the collaboration.

We describe our multi-queue architecture that provides support for these features and an implementation that uses prioritized interruptible-resumable independent and collaborative behaviors in a commercial game, BioWare Corp.'s *NWN*. We demonstrate how this architecture can support generative behaviors created by game designers who are non-programmers.

Behavior Architecture

The behavior architecture is based on a series of queues that support different kinds of behaviors. We begin by describing the ontology of the necessary behavior kinds.

An Ontology for Behaviors

All behaviors for an NPC are placed into a *role* that selects behaviors, and each behavior consists of a sequence of *tasks* or behaviors. We use a tavern that contains three kinds of NPCs: *Patron*, *Server,* and *Owner* to illustrate our behavior architecture. We assume the tavern contains one owner, a few servers, and many patrons. Figure 3 shows a subset of the behaviors for the *Patron* and one for the *Server*. We use a graphical representation that is reminiscent of both BTs and HSMs. Some of the behaviors (rectangles) in Figure 3 are expanded to show their tasks (ovals). The text in the clear rounded rectangles indicates some of the most important parameter values of the behaviors/tasks. For example, in Figure 3, the *Approach* behavior finds a *random creature* in the room.

When it is used, a behavior is tagged by two properties: *independent* or *collaborative*, as well as *proactive, latent*, or *reactive*. An *independent behavior* is performed alone. For example, in Figure 3, the *Approach* (*random creature*) behavior is independent since a *Patron* can *Approach* another creature without agreement by that creature. A *collaborative behavior* is performed jointly with another NPC (not PC). For example, the *Converse talk* behavior in Figure 3 is collaborative. The individual who starts a collaboration is called the *initiator* and all other collaborating individuals are called *reactors*.

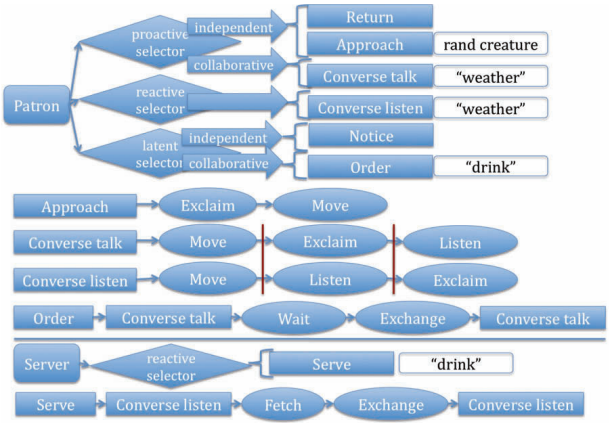

Figure 3 Some *Patron* behaviors and one *Server* behavior

Behavior initiation determines if a behavior is *proactive*, *latent,* or *reactive*. An independent or collaborative *proactive* behavior is spontaneously initiated by an NPC, when it has no active behaviors. We assume that roles select proactive behaviors probabilistically after filtering on any conditions they contain, but any of the behavior selection mechanisms listed in the introduction could be used. For example, the *Patron* in Figure 3 will initiate one of the *Return*, *Approach,* or *Converse talk* proactive behaviors whenever it must select a new behavior. A *latent* behavior is performed when an *event cue* fires. An *event cue* can be constructed from any game event such as a timer, a creature coming within some range, or a container being opened. An event cue can optionally contain pre-conditions. For example, the *Patron* in Figure 3 will perform an independent *Notice* (gaze shift) latent whenever another creature comes within a short range. The *Patron* will perform a collaborative *Order* latent whenever a *Server* comes close to the NPC and the NPC is thirsty. A *reactive* behavior is performed when an initiating NPC makes *eye-contact* (described later) with a potential collaborating NPC, so every reactive behavior is collaborative. For example, the *Patron* in Figure 3 can perform a *Converse listen* behavior on the topic of "weather" if another NPC tries to engage that NPC using a collaborative *Converse talk* behavior on the same topic.

Using Priorities to Guide Behavior Interruptions

Common game scenarios need priorities for designing highly responsive characters. For example, to be realistic, the latent *Notice* (gaze shift) behavior shown in Figure 3

should briefly interrupt most other behaviors. However, to simplify priorities for designers, we assign a set of default priorities. Each proactive independent behavior has a default priority of 0, each proactive collaborative behavior and reactive behavior has default priority 1, and each latent behavior has default priority 2. A latent should almost always interrupt a proactive. For example, the brief *Notice* interruption can occur while the *Patron* is performing any proactive or reactive behavior. Similarly, a thirsty *Patron* can opportunistically interrupt any proactive or reactive behavior to *Order* a drink when a *Server* passes by. Normally, a *Notice* will not interrupt an *Order* behavior, since that behavior is also latent with default priority 2. However, a designer can manually assign a higher priority. Note that realistic behaviors not only require interrupts, but also resumption of previous behaviors.

For example, a movie clip (www.cs.ualberta.ca/~script/movies/resumablebehavior.mov) shows *Patron Mary*, performing an *Approach* behavior towards *Patron Bob*. The behavior has two tasks, *Exclaim* and *Move*. The *Exclaim* task completes, but the *Move* task is interrupted by the *Notice* behavior when the PC gets close. *Notice* is just a glance, but it is implemented using an *Exclaim* task to emphasize it in the movie. After the *Notice* is complete, *Mary* resumes her *Approach* behavior, by re-starting the interrupted *Move* task (the *Exclaim* task of the *Approach* behavior is not performed again). The architectural mechanism for interrupting-resuming behaviors is discussed in the *Behavior Multi-queues* section.

Collaborative Behaviors

We limit collaboration to a pair of NPCs since our current implementation only supports pairs. However, there is nothing inherent in the architecture that limits collaboration to a pair. Each collaborative behavior has a topic parameter and two NPCs will collaborate if one of them (the *initiator*) has a proactive or latent behavior on a *topic* string and the other (the *reactor*) has a reactive behavior on the same topic. The same NPC can be both an initiator and a reactor on the same topic. For example, a *Patron* can initiate a conversation about the "weather" using a proactive collaborative *Converse talk* behavior and another *Patron* can react using the reactive *Converse listen* behavior on the "weather" topic. In fact, any other NPC that has a reactive behavior with topic "weather" could also react, even if that behavior is not a *Converse listen* behavior. The connection is made by topic, not by behavior. This allows a designer to add other behaviors on a topic at any time without editing the existing behaviors.

The potential reactor NPCs are filtered to include only those who satisfy three requirements. First, the NPC must have a reactive behavior on the same topic. Second, all conditions included in the initiator's collaborative behavior must be satisfied. Third, the reactor must make *eye-contact* with the initiator. *Eye-contact* is successful if the potential reactor is not currently involved in another active collaborative or latent behavior. Eye-contact is successful if the potential reactor is involved in an active independent behavior or if all of the reactor's collaborative and latent behaviors are suspended for some reason (described later). An independent behavior is interrupted and it resumes after the collaborative behavior is complete. For example, a *Patron* will only attempt to *Converse talk* with another NPC about the "weather" if that NPC has a reactive behavior whose topic is "weather", the *Patron* is within a certain range of this NPC, and the NPC is not already involved in a collaborative or a latent behavior.

For example, a movie clip (www.cs.ualberta.ca/~script/movies/collaborativebehavior.mov) shows *Patron Mary* performing an *Approach* behavior towards *Patron Bob*. Meanwhile, *Patron Dave* tries to make eye-contact to start a collaboration on topic "weather". Since *Mary*'s *Approach* behavior is independent, eye-contact is successful, the *Approach* behavior is interrupted, and *Mary* performs a reactive *Converse listen* behavior. When the conversation is done, *Mary* resumes her *Approach* behavior adjusting her trajectory, since *Bob* has moved.

When a collaboration begins, both NPCs start executing pairs of tasks that comprise their respective behaviors. For example, in Figure 3, *Move* is the common first task of both the *Converse talk* and *Converse listen* behaviors, so the NPCs move to each other. After both tasks complete, the collaboration enters the next phase. The second task of the *Converse talk* behavior is *Exclaim*, and the second task of the *Converse listen* behavior is *Listen*, so the first collaborator *Exclaims* (talks and gestures) while the second collaborator *Listens* (gestures). The architecture inserts phase barriers between task pairs, as shown in Figure 3. For example, if the reactor completes the *Listen* task before the initiator completes the *Exclaim* task, the reactor waits for the initiator to finish before they proceed to the next task pair, *Listen* (*initiator*) and *Exclaim* (*reactor*).

Sometimes, it is convenient for the designer to use collaborative behaviors where the proactive part and reactive part have a different number of tasks. In this case, the architecture is responsible for adding an appropriate number of default (*Wait*) tasks to the end of the shorter behavior. For example, the designer could add a proactive behavior *Converse long* to an NPC, which is like *Converse talk*, but with an extra *Exclaim* task at the end. In this case, a reactive *Converse listen* on the same topic could be used to collaborate with a *Converse long* and, while the initiator is performing the extra *Exclaim* task, the reactor would simply play a *Wait* animation.

Behavior Multi-queues

The behavior multi-queue architecture has three sets of queues: (*proactive) independent*, *collaborative (proactive or reactive),* and *latent* (*independent*, *collaborative,* or *reactive*). There is only a single proactive independent queue, but there can be an arbitrary number of proactive collaborative and latent queues. When a behavior is created, its tasks are placed on one appropriate queue. The proactive independent queue can hold only a proactive independent behavior. A proactive collaborative queue can hold a proactive collaborative behavior or a reactive

behavior in response to a proactive collaborative behavior. A latent queue can hold an independent or collaborative latent behavior, or a reactive behavior that reacted to a latent behavior. Each queue knows the priority of its current behavior and a latent queue knows whether its behavior is independent or collaborative. Note that when we use the term *collaborative* queue, we refer to either a proactive collaborative queue or a latent queue that contains a collaborative behavior.

Each behavior (and its queue) in the multi-queue is either *active* or *suspended*. When a behavior is started, it is active. The term *current behavior* denotes the active behavior with the highest priority. The term *interrupt a behavior* means to immediately stop the currently executing task of that behavior (if there is one), without removing that task from its queue. The term *cancel a behavior* means to interrupt the behavior, clear all its tasks from its queue and, if it was collaborative, cancel the collaborating behavior. The term *suspend a behavior* means to prevent that behavior's next task from executing until the behavior is made active. The term *activate a behavior* means to interrupt the current behavior (if any) and to allow the next task of the activated behavior to start executing, if its behavior has highest priority, when the next *task start* signal is received by the dispatcher. Note that interrupting a behavior does not suspend it.

Behavior Dispatch

When an NPC is spawned, it starts performing a control loop that repeatedly sends a *timer signal* to its *behavior dispatcher*. However, this behavior dispatcher is also signaled by event cues and other game events. For example, when a task is complete, a *task done* signal is sent to the dispatcher. The *behavior dispatcher* handles these signals:

1. New behavior signal
2. Task done signal
3. Collaborator done signal
4. Phase done signal
5. Task start signal
6. Kill collaboration signal
7. Timer signal or previous kind of signal ignored

Case 1. *New behavior signal*: if the new behavior has lower priority than the current behavior, the signal is ignored and case 7 is performed. Otherwise, the current behavior is interrupted, the new behavior is queued on an appropriate (independent, collaborative, or latent) queue, and a task start signal is sent. If there is no empty appropriate queue, then the lowest priority appropriate behavior is cancelled and its queue is used for the new behavior. The new behavior signal is either: *latent*, *reactive,* or *proactive*. The behavior dispatcher sometimes generates a *new proactive signal* during case 5. If the new proactive is independent, the appropriate queue is independent. If the new proactive is collaborative, and eye-contact is not made, the signal is ignored and case 7 is performed. If eye-contact is made, a collaborative queue is used and a *new reactive* signal is sent to the collaborator, so that it can handle its corresponding reactive behavior. A *new reactive signal* is not generated in the behavior dispatcher of the reacting NPC. Instead, it is sent by the collaborating NPC when it performs a new collaborative behavior after eye-contact is made. If the initiating behavior was proactive, the appropriate queue is collaborative. If the initiating behavior was latent, a latent queue is used. A *new latent signal* is generated when a latent cue is triggered and the appropriate queue is latent.

Case 2. *Task done signal*: if the task is not collaborative, then the task is removed from its queue, a timeout clock is reset, and a new *task start* signal is generated. If the task is collaborative, then a *collaborator done* signal is sent to the collaborator. If the *collaborator done flag* is set, a *phase done* signal is sent; otherwise, a *self done flag* is set. These steps are necessary to ensure that both collaborators have completed a task before they both proceed to the next task.

Case 3. *Collaborator done signal*: if the *self done flag* is not already set, then set the *collaborator done flag* and return, since we must wait for our own task to be done. Otherwise, if the *self done flag* has been set, send a *phase done* signal to start the next phase. However, before sending this signal, if the behavior is suspended (because it is waiting for the collaborator), activate it.

Case 4. *Phase done signal*: de-queue the task, reset the timeout clock, and generate a *task start* signal.

Case 5. *Task start signal*: reset the timeout clock and start performing the first remaining task from the queue of the current behavior. If there is no current behavior (no active queues or all queues empty), create a new proactive behavior and send a new *proactive signal*.

Case 6. *Kill collaboration signal*: cancel the collaborative behavior and send a task start signal.

Case 7. *Timer signal or previous kind of signal ignored*: if the current behavior is not collaborative or if the timeout clock has not reached a wait time, do nothing. Otherwise, for a long wait time, cancel the behavior, send a *kill collaboration* signal to the collaborator, and send a *task start* signal or, if the medium wait time has occurred, suspend the current behavior and send a *task start* signal.

Behavior Scenario

In addition, to supporting explicit behavior interruption and resumption, the dispatcher allows self interruption of collaborative behaviors (Case 7), which enables an NPC to perform other behaviors while waiting for the collaborator to complete a lengthy task. A realistic complex scenario illustrates the power of this multi-queue architecture. The queues are shown in Figure 4 at a point in this scenario with annotations showing specific progress points depicted as consecutive integers in circles. We show only the six queues used in this scenario.

Patron Mary performs the independent priority 0 *Approach* behavior towards *Patron Bob*. Mary finishes her *Exclaim* task and starts her *Move* task. *Server Linda* passes close to *Mary*, who has not had a drink for a while. *Mary* interrupts her *Approach* behavior to perform a *new latent* priority 2 collaborative *Order* behavior with the *Server*

(Case 1 – *new latent*). The *Approach* behavior is *interrupted*, which means that its current task (*Move*) stops executing, but it is not removed from its queue and it is not suspended. The scenario has reached point 1 in Figure 4.

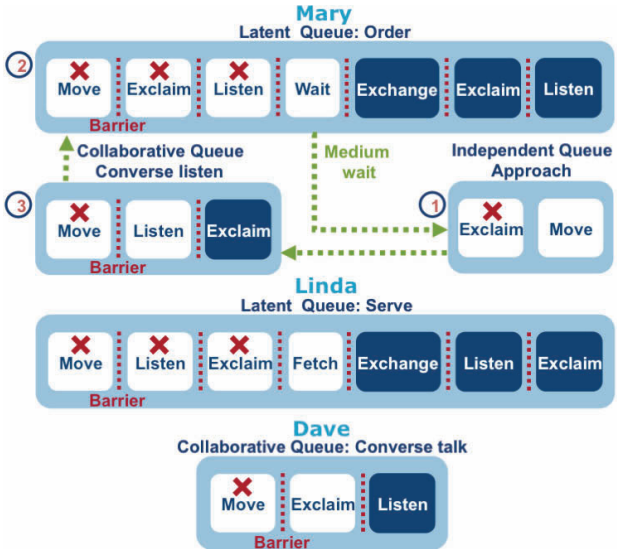

Figure 4 The multi-queues during a complex scenario

Even though the *Approach* behavior is not suspended, it has lower priority than the latent behavior, so no task from this behavior will execute until all higher priority behaviors are complete (Case 5). In performing an *Order*, *Mary* starts by performing a series of tasks: *Move*, *Exclaim*, *Listen*, *Wait* synchronized with *Linda*'s tasks: *Move*, *Listen*, *Exclaim*, *Fetch* and, at this point, Mary finishes the *Wait* task very quickly, while *Linda* performs *Fetch*. Since *Linda* takes a long time to fetch a drink, *Mary*'s dispatcher has a medium timeout (Case 7) and suspends the *Order* behavior. The dispatcher then re-performs the *Move* task of the *Approach* behavior (Case 5), which becomes the current behavior, since it is the only active behavior. The scenario has reached point 2 in Figure 4. After *Mary* takes a few steps toward *Bob*, *Patron Dave* tries to start a collaborative proactive *Converse talk* behavior with *Mary*. Since *Mary*'s only active behavior is independent, eye-contact is made and *Mary* starts the *Converse listen* reactive behavior (Case 1 – *new reactive*). Since *Dave*'s initiating behavior was proactive, the reactive *Converse listen* behavior uses a collaborative queue. Now *Linda* arrives with *Mary*'s drink order, completing the *Fetch* task. Since Mary's corresponding *Wait* task in the suspended *Order* behavior was marked as done, the *Order* behavior is activated (Case 3) and a *phase done* signal is sent. Activating a suspended behavior interrupts the current behavior (*Converse listen*). The scenario is at point 3 in Figure 4, just before the *phase done* signal is handled.

If the rest of the tasks in the priority 2 latent *Order* behavior are completed without interruption or timeout (*Exchange*/*Exchange*, *Move*/*Move*, *Exclaim*/*Listen*, *Listen*/*Exclaim*), then *Mary* will perform the *Listen* and *Exclaim* tasks in the priority 1 collaborative reactive *Converse listen* behavior, while Dave performs the synchronized tasks *Exclaim* and *Listen* in the *Converse talk* behavior. If they are completed without interruption, then the *Move* task in *Mary*'s priority 0 independent *Approach* behavior will finally be completed. This architecture supports interruptible and resumable behaviors. When a behavior resumes, it does not start over like the three stack behavior tree (Laming 2009). In addition, if a collaborative behavior waits for a very long time, it is cancelled and the corresponding behavior of its collaborator is cancelled. This is important if one of the collaborators is killed or otherwise unable to complete the collaboration.

Using and Designing Behaviors

To illustrate that our behavior architecture could be used in a generative manner by a non-programmer for a commercial game, we implemented a version of our multi-queue architecture in NWScript for BioWare's *NWN*. We used ScriptEase to create behavior patterns and implemented these patterns using our new architecture. Figure 5 shows the abbreviated *Patron* role behaviors from Figure 3 as expressed in ScriptEase.

Figure 5 The Patron tavern behaviors in ScriptEase

The proactive collaborative *Converse talk* behavior is open to show its three tasks: *Move*, *Exclaim*, and *Listen*. The *Exclaim* task is open to show the atomic actions that comprise this task. If the *Exclaim* task is interrupted, it re-starts from its first action, *faces* (highlighted).

In ScriptEase, we implemented proactive, reactive, and latent behaviors whose purposes are clear to the game designer. In some sense, ScriptEase ambient and latent behaviors can be represented as FSM transitions. For example, in Figure 5 they can be voluntary-proactive (from *Approach* to *Converse talk*) or forced-latent by perception (from *Approach* to *Order* when the server walks near the patron). However, it is hard to infer this differentiation (voluntary vs. forced) from an FSM diagram. In contrast, the ScriptEase generative pattern abstraction is easy to understand and to use. In fact, Grade 10 high-school

students with no programming experience created interactive game adventures (without behavior patterns) in four and a half hours following a two-day training period on the use of the *NWN* game, the *Aurora Toolset*, and ScriptEase (Carbonaro et al. 2008). In a new experiment, high-school game designers successfully used our new behavior patterns in ScriptEase (Cutumisu 2009). Our pattern model shields game designers from manual scripting and the synchronization issues posed by collaborative behaviors, allowing them to concentrate on story construction. Designers can easily group, manage, and reuse hundreds of behavior patterns.

Conclusion

A movie clip (www.cs.ualberta.ca/~script/movies/tavern.mov) shows a tavern scene with one owner, two servers, and eighteen patrons. This scene runs in game at more than 30 frames per second, despite the high activity in the scene. We have run this scene for days without any noticeable stalling of behaviors or NPCs who stop performing their designated behaviors. This illustrates that the multi-queue approach is both efficient and robust enough for commercial computer games. We have not seen a need for more than two collaborative and two latent queues in the kind of behaviors we observe or envisage for story-based games in the near future. When more than two queues are required, the lowest priority behavior is cancelled and the game situation seems natural. We only needed to change the default behavior priority occasionally and only for latent queues. For example, a guard that uses a priority 2 latent behavior to *Warn* an approaching character may need to be interrupted by a priority 3 latent behavior to *Attack* if that character or another character gets very close or actually touches a guarded object.

Sometimes an author may want to use tasks that are uninterruptible, or tasks that cannot be resumed. We can accommodate the first situation by assigning very high priorities to latent cues to create behaviors that are non-interruptible. We do not currently support behaviors that the designer might want to be non-resumable, but this feature is straightforward and will be added.

Since our implementation uses NWN, we currently have the luxury of relying on the game engine to provide low-level parallelism such as simultaneous walking and talking. An author can simply insert the walk and talk actions into a task in a behavior and the game engine uses its own internal action mechanism to ensure basic time-sharing between the actions. The NPC starts walking and, since this action takes a long time to complete, the talk action overlaps its execution. Our AI architecture supports this directly since the talk behavior could be added on a different queue and could be performed simultaneously with behaviors stored on existing queues. It takes minutes to insert existing behaviors into a game and bind the behavior parameters. It takes about an hour to design and create a new behavior. We view this architecture as a natural evolution of behavior trees.

References

Badler, N., Webber, B., Becket, W., Geib, C., Moore, M., Pelachaud, C., Reich, B., and Stone, M. 1995. Planning and parallel transition networks: Animation's new frontiers. In *Proceedings of the Computer Graphics and Applications: Pacific Graphics*, 101-117.

Carbonaro, M., Cutumisu, M., Duff, H., Gillis, S., Onuczko, C., Siegel, J., Schaeffer, J., Schumacher, A., Szafron, D. and Waugh, K. 2008. Interactive story authoring: A viable form of creative expression for the classroom. *Computers and Education 51 (2)*, 687-707.

Champandard, A. 2007. Behavior trees for next-gen game AI. In *Proceedings of Game Developers Conference*, Lyon.

Champandard, A., Verweij, T. and Straatman, R. 2009. The AI for Killzone 2's multiplayer bots. In *Proceedings of Game Developers Conference*, Paris.

Cutumisu, M. 2009. Using behavior patterns to generate scripts for computer role-playing games. PhD Thesis.

Dyckhoff, M. 2007. Evolving Halo's behavior tree AI. Invited talk at *the Game Developers Conference*, Lyon. http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf (accessed 2009).

Fu, D., Houlette, R. and Ludwig, J. 2007. An AI modeling tool for designers and developers. In *Proceedings of IEEE Aerospace Conference*, 1-9.

Gorniak, P. and Davis I. 2007. SquadSmart: Hierarchical planning and coordinated plan execution for squads of characters. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 14-19.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231-274.

Houlette, R. and Fu, D. 2003. The ultimate guide to FSMs in games. *AI Game Programming Wisdom 2*, Charles River Media.

Isla, D. 2005. Handling complexity in the Halo 2 AI. In *Proceedings of Game Developers Conference*, San Francisco.

Laming, B. 2009. From the ground Up: AI architecture and design patterns. In *Proceedings of Game Developers Conference*, San Francisco.

Mateas, M. and Stern, A. 2003. Façade: An experiment in building a fully-realized interactive drama. In *Proceedings of Game Developers Conference*, San Jose.

Orkin, J. 2006. Three states and a plan: The A.I. of F.E.A.R. In *Proceedings of Games Developers Conference*, San Jose.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI)*, 206-214.

Schwab, B. 2008. Implementation walkthrough of a homegrown "abstract state machine" style system in a commercial sports game. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 145-148.