36

Breathing Life into Your Background Characters

David "Rez" Graham

36.1 Introduction

Background AI is an important component of any game that contains nonplayer characters, critters, and other background elements, as it adds to the believability and immersion of your game. For the purposes of this article, NPCs are background characters, as opposed to enemy agents or major, story-driven characters, although there's nothing stopping you from applying these same principles to other characters as well.

It's important to provide the illusion that these background NPCs have their own lives and personalities so that your game world comes to life. If you have static NPCs that don't look like they're busy with their lives, your world will feel stale. The catch is that since these are minor NPCs, they can't cost very much in terms of memory, CPU time, or development time.

The chosen technique must also be able to scale in the number of NPCs processed, the complexity of their AI, and how often they're processed since it's likely that NPC behavior and content will get scaled back as cuts to the development schedule need to be made. There are performance and memory concerns as well. As the game comes to a close, CPU and memory intensive systems will end up getting scaled back, and it's important that the chosen solution can easily be scaled in either direction.

36.2 Common Techniques

There are a number of common techniques that many games use when adding background NPCs to their games. One approach is to use looping idles, where the NPC loops a single animation over and over. You might have a blacksmith who does nothing but hammer away at a sword for the entire duration of the game, or perhaps you have an innkeeper who is constantly wiping down the counter. This is often paired with other NPCs running random walk cycles where they simply choose a random location, walk there, idle for a minute, and then do it all over again. Unfortunately, neither of these techniques looks very good, which is not surprising when you consider that old RPGs from the NES and SNES era did the exact same thing.

Another common pitfall is to swing the pendulum in the other direction and attempt to implement a deep, rich AI system. If you already have a complex AI system you use for major NPCs or enemies, it might make sense to reuse that system for background characters, but this is often a mistake. By their very nature, background characters are not the focus of the player. They are just there for flavor. A much simpler system that is faster and uses less memory is usually the better choice here, because it allows you to have more background NPCs, which is important for creating the illusion of a large city. Only seeing four or five NPCs wandering around is not enough to convince the player that he's in a city, but seeing 20 or 30 might be. The goal here is to get the best of both worlds and be able to scale your solution towards either direction.

36.3 Description of the Schedule System

Three important criteria for determining if a strategy for background AI is feasible are high performance, scalability, and ease of implementation. Performance is important because you will want to run dozens or even hundreds of NPCs at one time. Running complex scoring functions during updates is often too expensive to be a viable solution. Having a modular system that lets you swap out complex components for simple ones lets you throttle the performance of the system without having to rewrite anything. The final solution needs to be very low-risk from an implementation and maintenance point of view as well. It should be as data-driven as possible so that once it's built, the designers and animators should be able to do 90% of the work necessary to create as many background NPCs as they desire.

One way to meet all three criteria and still get reasonable-looking behavior is to use a schedule system. A schedule can be thought of as a black box where the input is the current time and the output is an action. Under the covers, these actions are really nothing more than "go here and run this animation." By itself, a single NPC will still be rather predictable and stale, but if you have a dozen or more NPCs all running different schedules, you suddenly have a living, breathing world. At its heart, a schedule is really just a set of time blocks where each block defines the behavior for an NPC executing that schedule. The system is a lightweight, data-driven, hierarchical state machine, where the top-level state selection is based on time.

We used a system similar to this one on *The Sims Medieval* to give designers the flexibility they were after. It's also been used successfully on a number of RPGs and simulation games, which is where this kind of system works well. It can be used on other types of

games, like first-person shooters, adventure games, etc., but you may have to make some modifications. We'll discuss this further later in the chapter.

36.4 Schedule Architecture

There are three key components to this schedule system: *schedules*, *schedule entries*, and *actions*. Together, these concepts form the core of the background AI system.

36.4.1 Schedules

Schedules are linked to NPCs with a pointer, an ID, or anything else that's appropriate to the system. The schedule is the top-level interface for accessing any schedule data on the NPC and manipulating the schedule during runtime. Schedules contain a number of schedule entries, each of which represents a slice of time (see the following). With this system, NPCs may only be in one schedule entry at a time, though it wouldn't be too difficult to allow schedule layering as long as the rules were simple enough. One example would be to allow NPCs to consider all possible actions for all schedules they were in. Another idea is to allow an NPC to have multiple schedules but only one active schedule at a time. On *The Sims Medieval*, we chose the latter method.

Schedules are responsible for moving to the appropriate schedule entry based on the current time. They are also responsible for calling into the currently running entry and returning the appropriate action for the NPC to perform. Any functionality of a running schedule that needs to be accessed by the rest of the game should go here.

In the example code that accompanies this article (available on the book's website, http://www.gameaipro.com), the schedule component is implemented with the `ScheduleInstance` and `ScheduleDefinition` classes. `ScheduleInstance` represents the runtime data for a schedule, such as which entry is currently active, while `ScheduleDefinition` represents the static data that never changes, such as the ID and which schedule entries are tied to that schedule.

36.4.2 Schedule Entries

A *schedule entry* is a single slice of time within the schedule. It manages the lifecycle of the schedule entry and determines when it's time to move to the next entry. Schedule entries also encapsulate the decision-making an NPC performs when it is looking for something to do. For this reason, schedule entries are typically implemented with a strategy pattern, or other design pattern that enables you to easily swap one entry for another [Gamma et al. 01].

Whenever an NPC is idle within a schedule, it calls into the schedule to find a new action. The schedule calls into the currently active schedule entry, which in turn runs the actual decision-making logic to find a new action for the NPC to perform. This is returned up to the NPC, which then performs the appropriate action. The function in the schedule entry for choosing a new action is a pure virtual function whose implementation is dependent on the specific subclass that's instantiated. Schedule entries are all interchangeable and conform to a single interface. The specific schedule entry that is instantiated is determined by the type set in the data, which is passed to a factory.

Defining different schedule entry subclasses and overriding the action chooser function gives you a lot of control over how complex you want your background NPC AI logic to be.

You could return a single action, or a random action from a list, or you could run through a bunch of complex scoring algorithms that weigh NPC desires and goals based on the current world state. This is exactly what *The Sims Medieval* does. Each schedule entry had a set of desires for the Sim to fulfill above and beyond their standard desires. For example, the Blacksmith might attempt to satisfy his "Be a Blacksmith" desire by working at the forge.

Schedule entry types can also be mixed and matched. Some schedule entries could return a single action while others might use a more complex form of scoring. Changing the decision-making strategy is as simple as changing the schedule data. You can then build a library of various decision-making strategies and let your designers choose the one they want for each situation. This is the power of a data-driven scheduling system.

In the example code that accompanies this article, the schedule entry system starts with the `ScheduleEntry` base class, which declares the pure virtual function `ChooseNewAction()` as the action chooser. This class, along with the sample subclasses, is defined in the ScheduleEntry.cpp file.

36.4.3 Actions and Object Ownership

Actions are the output of schedules and represent things that NPCs do. They usually define a target object, the duration of the action, maybe an animation, and any other bits of data necessary to execute the action. Actions are very game-dependent; each game will likely have its own way of making NPCs interact with the world. This part of the system is the interface into your world. The actions returned by the action chooser in the schedule entry should be in whatever format works for your game. On *The Sims Medieval*, we called them *interaction definitions*. They were objects that contained the necessary information to instantiate an interaction on a Sim.

It's usually not enough to have an NPC interact with a targeted object. It's much more common to tell an NPC to sleep in its own bed or to work at its particular forge. We want the best of both worlds, so the action system needs to deal with the concept of object ownership as well.

All game objects that can be interacted with should have a type ID, which defines the type of object it is. The object type is really just metadata for the designer to label groups of similar objects. For example, all beds could be grouped under the "bed" type. This allows NPCs to own an object of a particular type, which greatly simplifies the scheduling data. For example, a schedule could have an action that tells the NPC to go to bed. The NPC will look in its map of owned objects and check to see if it has a "bed" object. If it does, it will use that bed. If not, some default behavior can be defined. Perhaps the NPC will choose a random bed, or perhaps it will fail the action and choose a new one. This same action could be applied to multiple NPCs without modification and it would work just fine. Each NPC would go to its own appropriate bed.

Defining object ownership is also game-specific. One common approach is to set up object ownership in your NPC tool as a set of object type to object ID mappings. This way, it can be specified by designers. Another approach is to support larger, more encompassing tags. For example, on *The Sims Medieval* designers had a spreadsheet that allowed them to map NPCs to their home and work buildings. This gave them permission to use all of the objects in those buildings so when they were told to "go home," they went to the appropriate place.

As an example, consider the sample action XML definition in Listing 36.1.

Listing 36.1. This is an example of an XML definition for an action.

```
<Action name = "bake_in_oven" target = "baker_oven"
        minDuration = "0.1" maxDuration = "0.2"
        animation = "bake_bread.anm"/>
```

The target of the action is an object type ID. This is not a specific object in the world, but rather a type of object. This will cause the NPC to search its mapping of objects it owns and, if it owns the appropriate type of object, perform the action on that object. If it doesn't, it will run some sort of default behavior, such as finding the closest object of that type.

This also allows you to target a group where multiple NPCs may own different objects. In the example above, there may be multiple oven objects with the "baker_oven" type, each owned by a different NPC. This cuts down on data duplication quite a bit.

The system as described only allows NPCs to own a single object of any given type. As long as you have reasonable rules for handling it, there's nothing stopping you from allowing NPCs to own multiple objects of the same type. One example would be to always choose the closest object.

In the example code that accompanies this article, actions are split into instances and definitions, just like schedules. There's the `ActionInstance` class and the `ActionDefinition` class. The `ActionInstance` class handles all the runtime data while the `ActionDefinition` handles the static data.

36.4.4 Putting It All Together

Putting all three of these systems together gives you a great framework for creating a simple scheduling system. To illustrate the point further, take a look at Listing 36.2.

Listing 36.2. An example of an NPC schedule with two entries. The first entry is a simple sleep entry that causes the NPC to sleep in its bed. The second entry has two actions which are chosen from a weighted random distribution. The NPC has a 25% chance to work at the copier and a 75% chance to work at its desk.

```
<Schedule name = "Drone">
        <Entry name = "Sleep" type = "Simple" startTime = "22">
                <Action type = "sleep"/>
        </Entry>
        <Entry name = "work_at_desk" type = "WeightedRandom"
                        startTime = "8">
                <Action type = "work_at_desk" weight = "3"/>
                <Action type = "work_at_copier" weight = "1"/>
        </Entry>
</Schedule>
```

36.5 Schedule Templates

One downside of schedules is that they tend to result in a lot of data duplication. For example, if your game has a day/night cycle, it's likely every NPC will have a schedule entry for sleeping. In fact, many of the entries will be very similar with each other. You can mitigate this by allowing templates and data inheritance.

Let's say that you have a guard schedule entry that determines where a guard should stand. If you build this entry in a template, you can apply that template to every guard's schedule and override just the data that's specific to that particular schedule. Adding this functionality is pretty straightforward. The template itself should have the exact same schema as the schedule entry, so all you need to do is create a schedule entry for each template and store it in a table for easy lookup.

Take a look at Listing 36.3 for an example of what this template might look like in XML.

Listing 36.3. Two schedule entry templates, one for sleeping and one for guarding.

```
<ScheduleEntryTemplates>
        <Entry name = "SleepTemplate" type = "Simple">
                <Action type = "sleep"/>
        </Entry>
        <Entry name = "GuardTemplate" type = "WeightedRandom">
                <!-- prefer to guard the residential area -->
                <Action type = "guard_west" weight = "2"/>
                <Action type = "guard_east" weight = "1"/>
        </Entry>
</ScheduleEntryTemplates>
```

Defining the guard schedules is just a matter of pointing to the templates and overriding the values you care about, as shown in Listing 36.4.

Listing 36.4. Two schedules using the same templates with overrides.

```
<Schedule name = "DayGuard">
        <Entry name = "Sleep" template = "SleepTemplate"
                startTime = "16"/>
        <Entry name = "StandGuard" template = "GuardTemplate"
                startTime = "8"/>
</Schedule>
<Schedule name = "NightGuard">
        <Entry name = "Sleep" template = "SleepTemplate"
                startTime = "8"/>
        <Entry name = "StandGuard" template = "GuardTemplate"
                startTime = "16"/>
</Schedule>
```

36.6 Improvements

There are a number of improvements that can be made to this system. First, only schedule entries use the template system, when in fact all of the schedule components could benefit from it. Every part of a schedule should be inheritable so that designers can build up a toolbox of different schedule parts to compose into final schedules for their NPCs.

Another improvement is in the handling of schedule caching. Currently, every schedule, template, action, and entry is stored in memory. This was done for simplicity and ease of explanation. A more appropriate implementation will depend on the design of your game. If your NPCs rarely switch schedules, you can flush out the schedules that are currently not being used and load them dynamically when necessary. This could potentially even be done on the schedule entry layer, since moving from one entry to another doesn't happen very often.

In the demo, IDs are handled as strings. This is how schedules are linked to actions and NPCs. This is another place where we chose this method due to its simplicity and ease of explanation. Every game uses something different for object IDs and has its own method of linking objects together. However, it's worth noting that doing string compares to match IDs is not a good strategy. At the very least, you'll want to hash those strings into unique IDs.

Another thing to consider is the rigid nature of schedules. If you have a schedule that sends NPCs to an inn for dinner and drinks at 6:00pm every day, you could easily have a pile of NPCs all stopping whatever they're doing and immediately heading to the inn at the exact same time. A better solution is to randomize the schedule update time with a Gaussian distribution function centered on the end time for that schedule entry. This will cause NPCs to move to their next entry at a time reasonably close to the tuned time, as long as you don't mind them being a little early sometimes. You can also rely on the action itself to inject some randomization. This is what we did on *The Sims Medieval*. Sims who changed to a new schedule would finish their current interaction before going to their scheduled location.

Integrating a scheduling system into your existing AI system can be very powerful. The core AI system needs to be able to override the schedule system when necessary. For example, if the player attacks the baker, the baker's schedule should be thrown out the window and he should either fight back or run away. Once the threat is gone, he can resume his schedule. This can work for enemies as well. You could assign a schedule to all enemies that will get overridden as soon as they notice the player.

Right now, schedule entries are defined by time. They represent what an NPC should be doing at a particular time, but not all games have the concept of time. Fortunately, it's easy to change this to anything else. For example, you could have schedules that choose schedule entries based on game events or player movements. You could have NPCs exist in a guarding schedule entry that changes when an alarm is raised.

You could even mix the two and have some schedule entries that are controlled by time and others that are controlled by events. This would allow you to have NPCs that run around on schedules but can also react to the world around them. For example, you could have a shopkeeper with a typical work schedule who can also respond when the town is attacked by orcs. Remember, the underlying structure behind this schedule system is a hierarchical finite-state machine. It's relatively easy to bring it closer to its roots.

Another limitation of the current system is that NPCs can only own a single object of any given type. As long as you have reasonable rules, there's nothing stopping you from allowing NPCs to own multiple objects of the same type.

Finally, all of the schedule data is in XML. Asking a designer to hand-edit this XML is not very realistic, so you will need to build a scheduling tool that can generate the necessary data and link it to game objects. A simple C# application with a grid control should suffice.

Note that this is a place where having a component-based object hierarchy [Rene 05] can really help. You can simply build a component that handles all the schedule data and attach it to any game object that needs to be controlled through a schedule. You could have another component that tracks the object type and allows objects with that component to be targets for schedules. This would allow you to easily slot in the schedule system.

36.7 Conclusion

NPCs can make or break the player's suspension of disbelief in your game. If you have an NPC that spends the entire game nailing in a board on a wall or polishing a sword, the illusion of a living world starts to go out the window. Fortunately, building believable background characters can be achieved quickly and easily using a scheduling system. NPCs are locked into schedules and updated with very simple behavioral AI. With just a handful of schedules, the hustle and bustle of a town will begin to emerge. Players will see NPCs going about their regular day and begin to infer a story from them. With a very small amount of work, your game worlds can come alive.

References

[Gamma et al. 01] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, pp. 315–323.

[Rene 05] B. Rene. "Component based object management." In *Game Programming Gems 5*, edited by Kim Pallister. Reading, MA: Charles River Media, 2005, pp. 25–37.