Chapter 9: Non-player characters

(Movement • Other NPC actions • Taking orders • Ask and tell • Choice-based conversation)

Animated non-player characters (NPCs) can really make the game world come alive. In their simplest form, NPCs are just ordinary objects whose antics are narrated spontaneously from time to time:

```
#field
(name *)  field
```

```
#field
(name *) field
(room *)
(look *)
        A scarecrow is here.
(on every tick in *)
        (par)
        (select)
        The scarecrow looks wistfully towards the horizon.
        (or)
        A growling noise emerges from the scarecrow's stomach.
        (or)
        The scarecrow suddenly sneezes.
        (or)
        (or)
        (or)
        (at random)

#scarecrow
(name *) scarecrow
(* is #in #field)
```

[Copy to clipboard]

A slightly more sophisticated NPC might interact with other objects in the room, such as the player's possessions:

```
#alley
(name *)  back alley
```

```
#alley
(name *) back alley
(room *)
(look *)
        A suspicious-looking figure is lurking in the shadows.
(on every tick in *)
        (current player $Player)
        (collect $Obj)
        *($Obj is #heldby $Player)
        (into $List)
        %% The following rule fails (which is fine) if the list is empty.
        (randomly select $Target from $List)
        (par)
        The suspicious-looking figure eyes (the $Target) with interest.
        (random from 1 to 5 into 1)    %% Fails 4 times out of 5.
        (now) ($Target is #heldby #thief)

#thief
(name *) suspicious-looking figure
(dict *)    suspicious looking thief
(animate *)
(* is #in #alley)
(descr *)
        (if) ($ is #heldby *) (then)
        The figure seems to be carrying something.
        (else)
```

```
        Very suspicious-looking.
        (endif)
```
[Copy to clipboard]

The library predicate (randomly select $Element from $List) picks a random element from a given list, or fails if the list is empty.

Movement

The library provides a predicate called (let $NPC go $Direction), to allow NPCs to roam the game world.

Here's a character who moves around on a fixed schedule:

```
#workshop
(name *)  workshop
```

```
#workshop
(name *) workshop
(room *)
(look *)   You're in a noisy workshop.
(from * go #east to #backroom)

#backroom
(name *) back room
(room *)
(look *)   Shelves line the walls in this dimly lit room.
(from * go #west to #workshop)

#mechanic
(name *) mechanic
(female *)
(appearance * $ $)
        A busy-looking mechanic is here, looking busy.
(* is #in #workshop)

(on every tick)
        (par)
        (select)
        (or)
        (or)
        (let * go #east)
        (or)
        (or)
        (let * go #west)
        (cycling)
```
[Copy to clipboard]

The caller of (let $ go $)—typically an (on every tick) rule, as above—is responsible for supplying a direction that corresponds to a valid exit.

It is instructive to look at the library code for (let $ go $):

```
(let $NPC go $Dir)
      ($NPC is in
```

```
(let $NPC go $Dir)
 ($NPC is in room $OldRoom)
 (from $OldRoom go $Dir to room $NewRoom)
 (if) (player can see $NPC) (then)
 (narrate $NPC leaving $OldRoom $Dir to $NewRoom)
 (endif)
 (now) ($NPC is #in $NewRoom)
 (if) (player can see $NPC) (then)
 (narrate $NPC entering $NewRoom from $OldRoom)
 (endif)
```
[Copy to clipboard]

The predicate ($ is in room $) traverses the object tree, via ($ has parent $) links, from a given object all the way to its enclosing room.

The predicate (from $ go $ to room $) consults the story-supplied tables of obvious exits and doors, (from $ go $ to $) and (from $ through $ to $), to find out what room lies in a particular direction. It is also possible to query this predicate “backwards” with a given originating room and neighbouring room, to obtain the direction.

The library provides default implementations of (narrate $ leaving $ $ to $) and (narrate $ entering $ from $), deliberately bland, which the story author may wish to override for flavour.

Note in particular the queries to (player can see $). That predicate also comes in handy when writing story code that's dealing with moving NPCs, because we generally only want to describe their actions when they're in the same room as the player:

```
#mechanic
```

```
#mechanic
(on every tick)
 (par)
 (select)
 (if) (player can see *) (then)
 The mechanic
 (select)
 operates the hydraulic drill.
 (or)
 removes a couple of tiny screws.
 (or)
 makes a precise measurement.
 (at random)
 (endif)
 (or)
 (let * go #east)
 (or)
 (if) (player can see *) (then)
 The mechanic
 (select)
 rifles through a box of parts.
 (or)
 looks something up in a binder.
 (or)
 wipes her hands on a dirty cloth.
 (at random)
 (endif)
 (or)
 (let * go #west)
 (cycling)
```

[Copy to clipboard]

Random movement

Here is an example of an NPC moving randomly through the game world:

```
#rover
(name *)  rover
```

```
#rover
(name *) rover
(singleton *)
(on every tick)
        (par)
        (* is in room $Room)
        (collect $Dir)
        *(from $Room go $Dir to room $)
        (into $Exits)
        (randomly select $Dir from $Exits)
        (let * go $Dir)
```

[Copy to clipboard]

But in the above example, the rover might easily go back and forth between neighbouring rooms several times in a row. To avoid that, we can refine the code with a global variable that keeps track of the last direction of movement—or rather its opposite:

```
#rover
(name *)  rover
```

```
#rover
(name *) rover
(singleton *)
(global variable (rover avoids direction $))
(on every tick)
        (par)
        (* is in room $Room)
        (collect $Dir)
        *(from $Room go $Dir to room $)
        ~(rover avoids direction $Dir)
        (into $Exits)
        %% Clear the variable so we don't get stuck in dead-end rooms:
        (now) ~(rover avoids direction $)
        (randomly select $Dir from $Exits)    %% This can fail.
        (let * go $Dir)
        (opposite of $Dir is $OppDir)
        (now) (rover avoids direction $OppDir)
```

[Copy to clipboard]

To populate the game world with several identical rovers, we'd have to animate each one of them. In that case, it would make sense to use a per-object variable instead of a global variable:

```
#redrover
(name *)  red rover
```

```
#redrover
(name *) red rover
#greenrover
(name *) green rover
#bluerover
(name *) blue rover
(on every tick)
        *($Rover is one of [#redrover #greenrover #bluerover])
        (par)
        ($Rover is in room $Room)
        (collect $Dir)
        *(from $Room go $Dir to room $)
        ~($Rover avoids direction $Dir)
        (into $Exits)
        %% Clear the variable so we don't get stuck in dead-end rooms:
        (now) ~($Rover avoids direction $)
        (randomly select $Dir from $Exits)
        (let $Rover go $Dir)
        (opposite of $Dir is $OppDir)
        (now) ($Rover avoids direction $OppDir)
```

[Copy to clipboard]

Moving towards an object

Using the path-finder from the standard library, it's straightforward to create an NPC that moves towards another object, such as the player character:

```
#duckling
(name *)  duckling
```

```
#duckling
(name *) duckling
(animate *)
```

```
(on every tick)
        %% Only do this with 50% probability, to allow the player to get away.
        (random from 1 to 2 into 1)
        (par)
        (current player $Player)
        ($Player is in room $Target)
        (* is in room $Here)
        (if) (first step from $Here to $Target is $Dir) (then)
        (let * go $Dir)
        (elseif) (player can see *) (then)
        “Quack.”
        (endif)
```

[Copy to clipboard]

Other NPC actions

In the previous section, we saw that (let $ go $) can be used to move non-player characters around, and that it makes queries to (narrate $ leaving $ $ to $) and (narrate $ entering $ from $).

The following predicates are also provided:

Let-predicate	Narration predicate
(let $NPC take $Obj)	(narrate $NPC taking $Obj)
(let $NPC drop $Obj)	(narrate $NPC dropping $Obj)
(let $NPC wear $Obj)	(narrate $NPC wearing $Obj)
(let $NPC remove $Obj)	(narrate $NPC removing $Obj)
(let $NPC put $A $Rel $B)	(narrate $NPC putting $A $Rel $B)
(let $NPC open $Obj)	(narrate $NPC opening $Obj)
(let $NPC close $Obj)	(narrate $NPC closing $Obj)
(let $NPC climb $Obj)	(narrate $NPC climbing $Obj)
(let $NPC enter $Obj)	(narrate $NPC entering $Obj)
(let $NPC leave $Obj)	(narrate $NPC leaving $Obj)

All of these let-predicates have the same internal structure: First they check whether the player can see the action, and query the narration predicate if so. Then they update the world model, which often boils down to a single (now) statement.

In other words, these predicates are quite simplistic. They can work as a foundation for more sophisticated story-specific rules, or as a template for rapid prototyping of game ideas. But it is also possible to sidestep them entirely, and do everything from within the (on every tick) rules.

Note in particular that these predicates do not check whether the requested NPC action is possible; that is the responsibility of the story author. If you let an NPC pick up the moon, they will happily go ahead and do so, even if it's in a different room and not an item.

Taking orders

The parser understands e.g. TELL BOB TO GO EAST as well as BOB, E as the action [tell #bob to go #east]. That is, [tell $NPC to | $Action]—a list of the word tell, the NPC object, the word to, followed by whatever elements make up the requested action.

By default, NPCs refuse all such requests. Obedience is enabled on a case-by-case basis, by overriding action-handling predicates as usual. We can make use of the let-predicates of the previous section, but remember that we have to ensure that the action is possible. Thus:

```
#bob
(name *)  Bob
```

```
#bob
(name *) Bob
(proper *)
(male *)

(perform [tell * to go $Dir])
        (* is in room $Room)
        (from $Room go $Dir to room $)

        (let * go $Dir)
```

[Copy to clipboard]

If there is no exit in the given direction, (from $ go $ to room $) fails, and the perform rule falls back on the default rule provided by the library, wherein Bob refuses.

The above is sufficient in a game where Bob is free to roam the map using obvious exits. But if Bob is e.g. locked up in a cage at some point, the rules have to take that into account. Thus, for instance:

```
#bob
(name *)  Bob
```

```
#bob
(name *) Bob
(proper *)
(male *)
(prevent [tell * to go $])
        (* is #in #cage)
        Bob scratches his head with the banana, and makes a sad gesture towards the cage door.
(perform [tell * to go $Dir])
        (* is in room $Room)
        (from $Room go $Dir to room $)
        “Oh oh. Ah ah!”
        (let * go $Dir)
```

[Copy to clipboard]

Ask and tell

The library provides a set of standard actions for communicating with NPCs. These include the classical “ask *x* about *y*” and “tell *x* about *y*” actions, which redirect by default via “talk to *x* about *y*” to a simple “talk to *x*”.

The story author deals with the [talk to $] action as with any other; namely by overriding the default action-handling predicates. For instance:

```
#librarian
(name *)  librarian
```

```
#librarian
(name *) librarian
(male *)
(perform [talk to *])
        “Shh! This is a library.”
```

[Copy to clipboard]

The ASK and TELL verbs redirect to a common TALK TO action by default, but they can be overridden with more specific responses. However, modern players may expect games to treat ASK, TELL, and TALK TO as synonyms.

```
#librarian
(name *)  librarian
```

```
#librarian
(name *) librarian
(male *)
(perform [talk to * about #book])
        “I'd like to borrow this book.” (line)
        “Certainly. May I see your library card, please?”
(perform [talk to * about $])
        (The *) has nothing to say about that.
```

[Copy to clipboard]

In the above example, ASK LIBRARIAN ABOUT BOOK only works if both the librarian and the the book are in scope. Otherwise, the game will print a generic “I don't understand” message.

But in text adventures—and real life—people often want to talk about things that are outside the current room. The library provides a mechanism for this: Objects with the trait (topic $) will always be recognized as valid topics, regardless of scope:

```
#fountain
(name *)  oddly-
```

```
#fountain
(name *) oddly-shaped fountain
```

```
(an *)
(* is #in #townsquare)
(topic *)
(perform [talk to #librarian about *])
        “What's up with the oddly-shaped fountain in the town square?” (line)
        (select)
        “Ah, yes. Old Chancellor Fhtagn. May I recommend this excellent
        introduction to the history of our town?” (line)
        The librarian hands you a dusty tome.
        (now) (#historybook is #heldby #player)
        (or)
        “It's Old Chancellor Fhtagn.”
        (stopping)
```
[Copy to clipboard]

Topic objects can also be used to refer to abstract concepts. Such intangible topics won't appear in any rooms, but they can have (name $) and (dict $) entries like any other objects. They are usually proper nouns. The game might refer to them during disambiguation: “Did you want to ask the librarian about life, the universe, and everything or the copy of Life magazine?”

```
#lifetopic
(name *)  life, the
```

```
#lifetopic
(name *) life, the universe, and everything
(proper *)
(topic *)
(perform [talk to #librarian about *])
        “That would be under 823.9, Modern Period Fiction. A for Adams.”
```
[Copy to clipboard]

As a convenience, (proper topic $) can be used to declare both traits in one go:

```
#lifetopic
(name *)  life, the
```

```
#lifetopic
(name *) life, the universe, and everything
(proper topic *)
```
[Copy to clipboard]

Using dictionary words to represent topics

To save memory, it is also possible to represent conversational topics using dictionary words.

A couple of rules such as these:

```
(topic keyword
@childhood)
```

```
(topic keyword @childhood)
(topic keyword @youth implies @childhood)
(describe topic @childhood)
  your childhood
```
[Copy to clipboard]

will make the parser understand TELL DOCTOR ABOUT THE SHATTERED DREAMS OF CHILDHOOD as [tell #doctor about childhood] (recall that the @ character can be omitted inside list expressions). But if you also define:

```
(topic keyword
@dreams)
```

```
(topic keyword @dreams)
(describe topic @dreams)
  your dreams
```
[Copy to clipboard]

then THE SHATTERED DREAMS OF CHILDHOOD contains two valid keywords, and the game will ask, did you mean to tell the doctor about your childhood or your dreams?

Looking for keywords is a simple approach to topic parsing, but it is occasionally too crude. In the next chapter, we will see how to write arbitrarily complex parser rules for topics.

The unrecognized topic

The dictionary word @? (a single question mark) represents a topic that was unrecognized by the parser. It can be handled as a special case:

```
(perform [talk to
#librarian about ?])
```

```
(perform [talk to #librarian about ?])
  "I really don't know about that."
```

[Copy to clipboard]

But bear in mind that the player might ask an NPC about any portable object, as well as any topic object. The parser will recognize those, and construct corresponding actions (with objects rather than @?), even if there are no explicit rule definitions that match those actions.

Most talking NPCs will therefore have some kind of catch-all rule, defined towards the end of the story source code, where they confess that they don't know much about the subject:

```
%% The following
must come after the
```

```
%% The following must come after the more specific [talk to #librarian about ...] rules.
(perform [talk to #librarian about $])
  "I really don't know about that."
```

[Copy to clipboard]

The two techniques above can be combined:

```
(perform [talk to
#librarian about ?])
```

```
(perform [talk to #librarian about ?])
  "I really don't know about that."
%% The following must come after the more specific [talk to #librarian about ...] rules.
(perform [talk to #librarian about $Obj])
  "I really don't know much about (the $Obj)."
```

[Copy to clipboard]

Choice-based conversation

One possible approach to conversation in parser games is to temporarily switch to choice-based interaction. If you haven't already read the chapter on choice mode, please do so before proceeding.

Typically, one would create a set of choice nodes where the labels represent lines spoken by the player character, and the display-texts contain responses from the NPC.

In the following playable example, the NPC object itself acts as a central hub node, offering an initial set of choices. Some choices cause the conversation to branch away from the hub in order to focus on a particular subject, represented by a separate hub-like structure. Some of the choices are gated, and appear only if certain other nodes of the conversation have been exposed.

```
#player
(current player *)
```

```
#player
(current player *)
(* is #in #repairshop)

#repairshop
(name *)        repair shop
(room *)
(look *)        You're in a noisy workshop.

#mechanic
(name *)        (if) (#mech-name is exposed) (then)
                Lisa
                (else)
```

busy-looking mechanic
(endif)
(proper *) (#mech-name is exposed)
(female *)
(* is #in #repairshop)
(appearance * $ $)
(A *) is here, looking busy.
(on every tick)
(if) (player can see *) (then)
(The *)
(select)
rifles through a box of parts.
(or)
removes a couple of screws.
(or)
checks the oil pressure of a clunker.
(at random)
(endif)
(perform [talk to *])
“Um, excuse me?”
(par)
“Can I help you?”
(The #mechanic) wipes her hands on a dirty cloth and turns to face you.
(activate node *)
(after disp (terminating $))
(par)
(try [look])
(tick)
#mech-nice
(#mechanic offers *)
(label *) “This looks like a nice establishment. Very authentic-looking.”
(disp *) “Happy to hear it.”
#mech-wrong
(#mechanic offers *)
(sticky *)
(initial label *) “There's something wrong with my car.”
(label *) “About my car again...”
(disp *) “Yee-es?”
(* flows to #mech-car)
#mech-car-nowork
(#mech-car offers *)
(label *) “It doesn't start anymore, is the thing.”
(disp *) “I see. Have you checked the battery?”
#mech-battery
(#mech-car offers *)
(#mech-car-nowork is exposed)
(label *) “How do I check the battery?”
(disp *) “Under the hood. Voltmeter on the plus and minus terminals.”
#mech-voltmeter
(#mech-car offers *)
(#mech-battery is exposed)
(label *) “Uh, what's a voltmeter?”
(disp *) “Or a multimeter. I could have a look at it, I guess. I have some time
next Thursday.”
#mech-gas
(#mech-car offers *)
(#mech-car-nowork is exposed)
(* is unexposed) %% Only offer this node once, even if it's not a dead end.
(label *) “The battery is brand new. Could it be something else, do you think?”
(disp *) “And you're sure that there's gas in the tank?”

```
#mech-gasyes

(#mech-gas offers *)
(label *)       “Gas? Oh, gas! Yes, the man who sold me the car specifically said
                there was gas in the tank.”
(disp *)        “I was afraid of that. This'll be expensive. I have a free timeslot next
                Thursday.”
(* flows to #mech-car)

#mech-gasno
(#mech-gas offers *)
(label *)       “I should certainly think not! That sounds positively dangerous.”
(disp *)        “I see. Well, I could have a look at it next Thursday.”
(* flows to #mech-car)

#mech-deal
(#mech-car offers *)
                (#mech-voltmeter is exposed) (or) (#mech-gas is exposed)
(label *)       “Next Thursday is fine.”
(disp *)        “All right then. You can leave the car out front.”
                (The #mechanic) returns to her work.
                (par)
                “Right. Goodbye for now, then!”
                (par)
                (The #mechanic) nods, and you walk out into the rain.
                (game over { You have no car. })

#mech-nevermind
(#mech-car offers *)
(sticky *)
(label *)       “Never mind.”
(* flows to #mechanic)

#mech-name
(#mechanic offers *)
(label *)       “What's your name?”
(disp *)        “I'm Lisa. Pleasure to meet you.”

#mech-bye
(#mechanic offers *)
(label *)       “It was nice talking to you!”
(disp *)        “Any time!” (The #mechanic) returns to her work.
(terminating *)
```
[Copy to clipboard]

And this is what it looks like:

Repair shop
You're in a noisy workshop.

A busy-looking mechanic is here, looking busy.

> talk to mechanic
“Um, excuse me?”

“Can I help you?” The busy-looking mechanic wipes her hands on a dirty cloth and turns to face you.

1. “This looks like a nice establishment. Very authentic-looking.”
2. “There's something wrong with my car.”
3. “What's your name?”
4. “It was nice talking to you!”
> 1
“This looks like a nice establishment. Very authentic-looking.”

“Happy to hear it.”

1. “There's something wrong with my car.”
2. “What's your name?”
3. “It was nice talking to you!”
> undo
Undoing the last turn.

Repair shop

1. “This looks like a nice establishment. Very authentic-looking.”
2. “There's something wrong with my car.”
3. “What's your name?”
4. “It was nice talking to you!”

> 3
“What's your name?”

“I'm Lisa. Pleasure to meet you.”

1. “This looks like a nice establishment. Very authentic-looking.”
2. “There's something wrong with my car.”
3. “It was nice talking to you!”

> 3
“It was nice talking to you!”

“Any time!” Lisa returns to her work.

Repair shop
You're in a noisy workshop.

Lisa is here, looking busy.

Lisa checks the oil pressure of a clunker.

>

Ask-and-tell topic objects can serve as shortcuts to conversation nodes. Recall that (choose $) prints the label, i.e. the player character's line, in a separate paragraph before activating the node:

```
#car
(name *)  my car
```

```
#car
(name *) my car
(topic *)
(perform [ask #mechanic about *])
         (choose #mech-wrong)

#voltmeter
(name *) voltmeter
(topic *)
(perform [ask #mechanic about *])
         (choose #mech-voltmeter)
         %% This node is a dead end, so we will remain in parser mode.
```

[Copy to clipboard]

Onwards to “Chapter 10: Understanding player input” • Back to the Table of Contents