FIGURE 7.50: Its gaze remains on the user as it is zapped

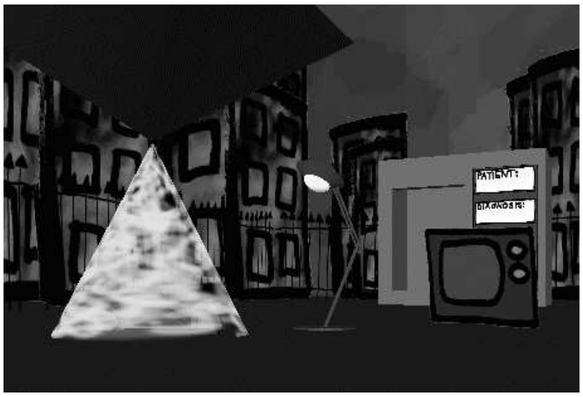

FIGURE 7.51: The End

is happening and that the machine is somehow involved. The Patient's glances at the camera seem to draw users into the scenario, generating a greater sense of connection with the scene and sympathy for the Patient's plight.

Transitions as Mindset Change

These case studies are only two examples of how transitions work; much more work needs to be done in order to explore how much of a difference they can make. But they do suggest that transitions change the qualitative perception of behavior by changing the nature of behavior from stimulus-response to reflection on the implications of what is happening around the agent. Transitions also change the way in which the designer tends to think, because they encourage the designer to think about and then make crystal-clear for the user the intended point of each behavioral change. This change in mindset ends up changing the nature of agent design in the Expressivator; we will explore these issues in the next section.

Agent Design in the Expressivator

Through its focus on transitions, the Expressivator changes the way designers must think about — and therefore go about — agent design. In Hap, the Expressivator's predecessor, an agent is defined in a number of steps:

1. Decide on the high-level behaviors in which the agent will engage.

2. Implement each high-level behavior, generally in terms of a number of low-level behaviors and some miscellaneous behavior to knit them together.

3. Use context-conditions, conflicts, and other design strategies to know when each behavior is appropriate for the creature to engage in.

The Expressivator more tightly constrains the agent design process. Similarly to Hap, the designer must first decide on a set of high-level signifiers the agent will express. But s/he must also decide on the transitions between the high-level signifiers; this includes deciding both which behaviors may lead to which other behaviors and the reasons the agent might want to make each behavioral switch. Similarly, for each high-level signifier, s/he must decompose it into a set of low-level signifiers and then explicitly decide how those low-level signifiers will interrelate.

Specifically, when building a high-level signifier, the agent designer must do the following:

1. Identify the low-level signifiers of which the high-level signifier is composed.

2. For each possible transition between low-level signifiers, determine the possible reasons for behavioral change (transition triggers).[3]

3. For each possible reason, determine how that reason should be communicated to the user (transition demons).

An example of such a design for the Patient is in Figure 7.52. Having made such a design, the intrepid agent builder must then implement each low-level signifier, transition trigger, and transition demon to create the high-level signifier.

Once the builder has engaged in this process for each high-level signifier, the high-level signifiers must be combined to form the complete agent. This involves a similar process of identifying each possible reason for each possible transition, and how each reason should be communicated. An example of this reasoning for transitions out of the "Turned Off" high-level signifier is in Figure 7.53.

In the end, the design of an agent involves two levels of atomization, with the atoms at each level interrelated through the use of transitions. The full structure of the Patient is shown in Figure 7.54.

A case study of the entire Patient design process is in Appendix C; this may interest humanists as well as technical readers.

[3] It turns out that quite a few of the theoretically possible transitions do not tend to make sense in practice. So this step is not quite as painful as one might expect.

High-level signifier: Mope by Fence
Low-level signifiers:

1. Look out at world
2. Sigh
3. Walk up and down fence

Relationships between low-level signifiers:

Behaviors	**Reason**	**How**
1→2	Life is bad! Wish I was out there!	Stop looking a moment Lost in reverie
1→3	Bored with spot. Get better position.	Look in the direction I am planning to walk. Focus on something there. Walk, keeping eye on spot.
2→1	I'm sad, but I still want to look	Interruption
3→1	Got to point where I can see the thing I want to look at	Turn to face and look at the thing intently

FIGURE 7.52: Design of the high-level signifier *Mope by Fence*.

New Behavior	**Reason**	**How**
Explore World Read Sign	Patient awakes from being turned off	Slowly rise up. Shake self. Blink, blink. Maybe sigh. Look around slowly to get orientation. This should be exagerrated the first time; after that it becomes a routine.
Exercise	Same reason	Here you should be exercising like a maniac while looking around for the Overseer. Taper off.
Mope by Fence	God, just another reason to be depressed	Same as transition to Explore World, but make it even more depressed.

FIGURE 7.53: The design for transitions out of Turned Off.

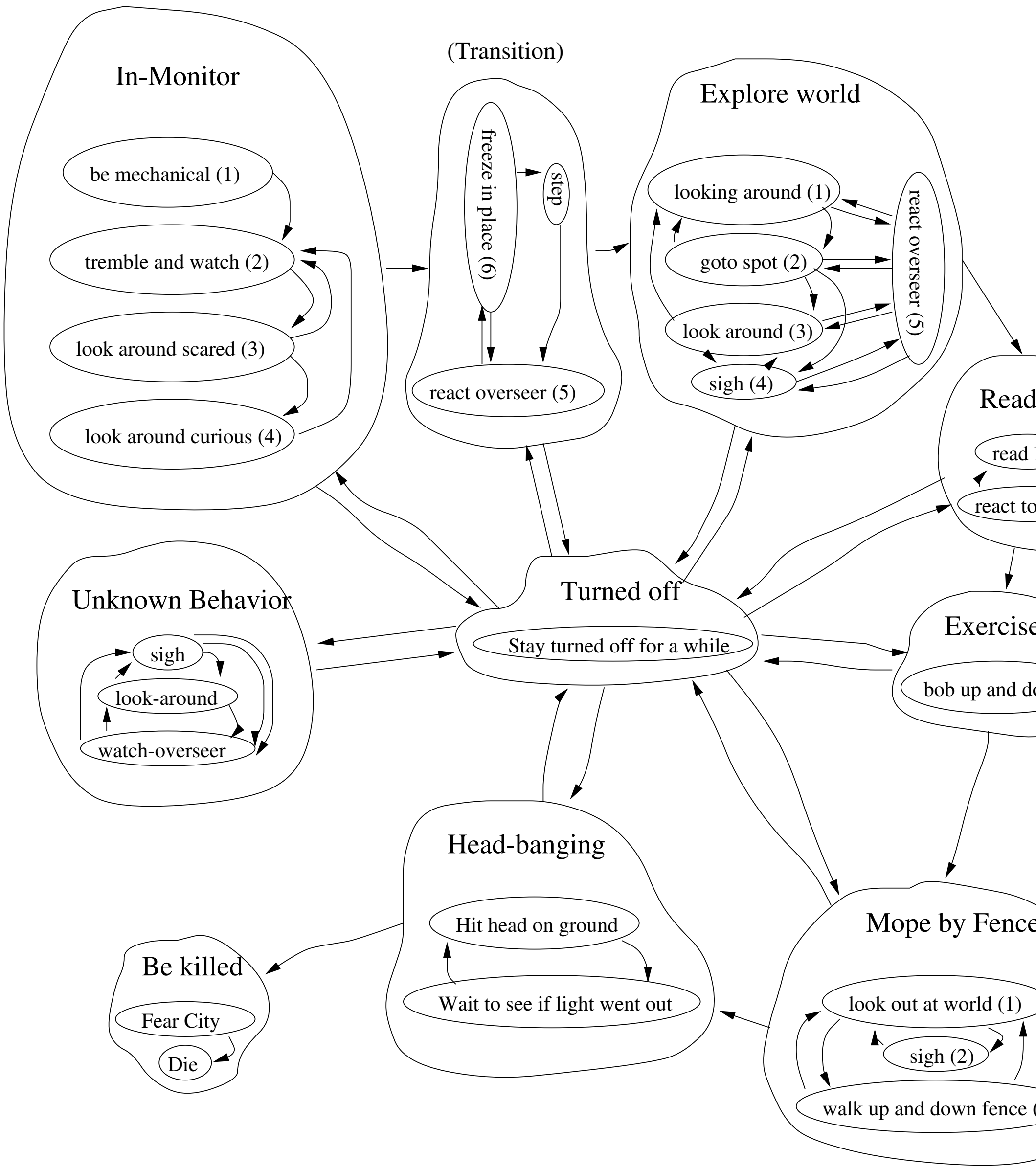

FIGURE 7.54: The complete design of the Patient, as implemented.

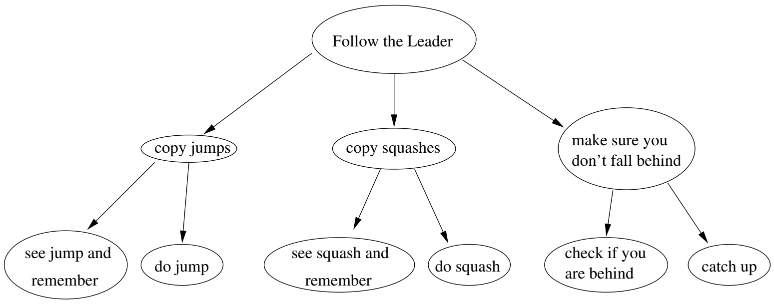

FIGURE 7.55: Follow the Leader structure in Hap

The Expressivator Mindset

In order to understand the Expressivator mindset more deeply, technical readers are suggested to take a moment now to read section D.2 of the Appendix.

The Expressivator changes the way the designer needs to think about behaviors. Because of the focus on transitions, the Expressivator demands that the designer know *why* the agent does what it does. In designing the Patient, I would many times want the agent to change behaviors, and discover to my surprise that I had no idea why the agent should change. I would be forced to stop and think about the reasons for the agent's behaviors; the articulation of those reasons would invariably clean up the behavior design.

But the change in mindset the Expressivator brings about goes deeper than this; it comes about from interactions between transitions and the redefinition of behaviors as signifiers. As discussed in Chapter 5, under the Expressivator framework behaviors are fundamentally things to be expressed to the audience. Complex behaviors may make an agent more intelligent, but if the audience cannot understand the complex nuances of the behavior, they are useless. Instead, under the Expressivator behaviors are simplified; the focus is on making them expressive. Instead of having complexity in the behaviors, *complexity comes from expressing to the user the interconnections between the behaviors*.

This change in mindset means behavioral code is structured differently. For example, when I worked on the Woggles, I built a behavior for following someone while playing the game Follow the Leader. The structure of this behavior is shown in Figure 7.55. The high-level behavior is broken up into three low-level behaviors, which all run simultaneously. Two behaviors are responsible for copying the leader's actions. One watches for the leader's "squash" actions, remembers how much the leader squashes, and squashes however the leader squashed last. The other watches for the leader's "jump" actions, remembers where the leader is jumping, and jumps wherever the leader went last. The third behavior is responsible for error recovery; it senses where the leader is, and if the leader is getting too far ahead, it takes over with a "catch-up" behavior that runs to where the leader is without bothering to copy the leader's actions.

The Follow the Leader behavior works well and robustly. The agent correctly follows what the leader does and is able to recover if the leader

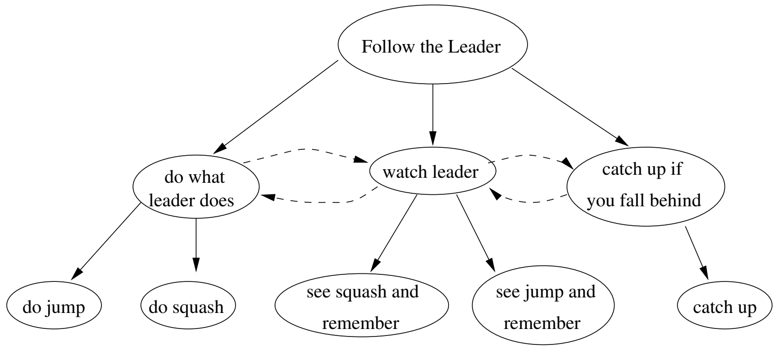

FIGURE 7.56: Follow the Leader structure in the Expressivator

From	**To**	**Reason**	**How**
Watch	Copy	Saw what the leader did	Turn glance from leader to where you are going
Copy	Watch	Want to know what leader does next	Pause; turn eyes to leader
Watch	Catch up	Can't see what leader is doing	Pause; strain to see leader; get nervous
Catch up	Watch	Caught up to leader	Pant; do subsequent behaviors more quickly

FIGURE 7.57: Transitions for Follow the Leader

is going too quickly for it. The flaw in it from the Expressivator's point of view is that the behavior is organized according to the logic of the activity, but *not according to what it is logical for the user to perceive*. We want to communicate to the user that the agent is watching the leader and copying its actions. But the actions of perception are split among all three behaviors and are generally done without corresponding movements of the agent's eyes; the action of copying is split into two completely independent behaviors. The Follow the Leader behavior is elegantly designed, but not optimal for communicating to the user what is going on.

Instead, a version of Follow the Leader for the Expressivator would require breaking up the activity of following into the things we would like the user to pick up on and their interrelationships. In Figure 7.56, we can see what such a structure might look like. Follow the Leader is now broken up into the behaviors that correspond what we want the user to notice: (1) watching the leader to find out what it is doing; (2) actually copying the leader's movements; and (3) catching up when the agent is behind. Each of these behaviors can be written relatively simply; the goal is not to do complex reasoning but to be sure to display clearly the basic idea of the behavior. Transitions (in dotted lines) are added to make the relationships between these behaviors clear; what these transitions mean is shown in Figure 7.57.

The heuristic of simplifying agent structure by focusing on its expressive aspects does not merely apply to the structure of the behaviors; it affects the entire design of the agent. For example, I built a rudimentary emotional system for the Patient (described in more detail in section A.2.5 of the Appendix). Originally, the Patient had a fear variable that would rise when the Overseer was near, and diminish when the Overseer went away. I then used the level of the fear variable to affect the Patient's behavior. The trouble with this system was that the Patient did not necessarily show any reaction to the Overseer's presence in conjunction with the change in fear. This means its fear would rise and fall without that fact being displayed to the user, making subsequent fearful behavior on the part of the Patient seem to come out of the blue. I therefore replaced this system with one where fear is increased whenever the Patient *visibly reacts* to the Overseer's presence. This model, where fear is the effect rather than the cause of fearful behavior, is psychologically dubious, but helps to ensure that users are kept apprised of the Patient's emotional situation.

Finally, it is not only the structure, but the content of behaviors that changes. Because the whole point of low-level signifiers is to communicate the agent's activity clearly to the user, most of the work in designing these signifiers is in working out the actual physical presentation of the behavior to the user. Rather than spending a lot of time on structuring code according to various conditions under which it should be engaged, the designer must spend substantial time with an animation package working on the details of motion. In some sense, the Expressivator reduces the problem of behavior generation to animation.

This emphasis on simple and extremely clear behavior contrasts with much current behavior-based AI work, in which the actual animated or robotic presentation of behaviors is considered trivial or beside the point. For the Expressivator, the level at which the basic units of meaning are communicated is essential; therefore, the graphical embodiment and manipulation of the agent, though perhaps not an "AI problem," is not a pleasant side-light but an essential part of what it means to be an agent.

Animation and Behavior-Based Programming: Battle of the Titans

The Expressivator demands that the agent designer spend substantial time getting the animated expression of the low-level signifiers right. I do not believe that I did this animation particularly well with the Industrial Graveyard. This is due to a number of reasons, starting with my continued subconscious inheritance of the AI concept that the code is the *real* agent and its graphical presentation only an afterthought. My lack of training as animator was another constant source of difficulty. But the major problem with creating adequate behavior for the Patient is that the substrate of the Expressivator, Hap, simply is not oriented to this way of thinking about agents.

I needed to use the Hap language in order to implement the low-level signifiers. Hap makes it easy (and fun!) to make complicated behavior with much variation based on the agent's mood, with reactions to any of a host of events that might be happening in the environment, with multiple processes running simultaneously, etc. What Hap does *not* do is make it easy to test and control low-level animation. This means it is relatively easy to build a dam-building behavior where a beaver searches for sticks and patches developing holes while keeping an eye out for predators; but it is relatively hard to create a sighing behavior that looks like sighing but not like panting or breathing. Getting the animation right in Hap is hard.

Humanists, don't worry if this paragraph is ununderstandable. Just move on to the next paragraph.

There are several difficulties with using Hap to generate animation. The most straightforward one is that Hap is a compiled language. That means in order to design a behavior one first writes a program, then one compiles it (a process that may take several minutes),[4] then one runs it. If the squashing looks just a touch too slow, one modifies the program, compiles it again, and runs it again. If now it is just a touch too fast, one goes through the whole procedure again. Every micro-change in the parameters of the code means several minutes of waiting before the designer can see the effect; the end effect is the designer feeling heartily encouraged not to fine-tune behavioral presentation.

But this problem is relatively easy to address. I did it by writing a Hap interpreter (which was itself written in Hap!). The interpreter would read in and execute new versions of the behavior while the simulation was running; low-level behaviors could now be tested and changed in the blink of an eye.

A more fundamental problem with Hap is the split it makes between the action architecture and the body. The gap between the actions that Hap produces and the actual movements the body ends up making as a result swallows up many fantasies of control of animated expression the agent designer may have.

Specifically, the agent's body is an articulated figure with 19 degrees of freedom, including such things as the body's position in space, the angle at which the agent holds its head, the body's color, and whether the body's light is on. As described in Chapter 5, rather than manipulating these parameters directly, Hap sends commands to the body at the level of actions; the motor system which receives these commands is then responsible for implementing the actions in a reasonable way given the physics of the world and other attributes of the body. For example, instead of telling the body to move to a particular point, Hap sends a command to "jump" to a particular point, reaching a particular height along its way. The motor system then calculates, based on the gravity of the world, what arc the jump should take and how much the agent should squash at the beginning and end of the jump. It also combines the physical manifestation of the jump correctly with actions that take place before and after the jump; so if the agent will continue into another jump, the motor system combines them so that the agent's momentum is carried through.

In this way, a single action generates numerous changes in the body's degrees of freedom over time in ways that depend on the agent's body, aspects of the environment, and the other actions that the agent has recently made or is about to make. This level of abstraction is essential because Hap is designed to control an agent in an uncertain environment, not generate pre-structured film clips. In essence, Hap sends down the

[4]To be precise, one compiles three times: once to turn the Hap into RAL code, once to turn the RAL into C code, and once to turn C into machine code.

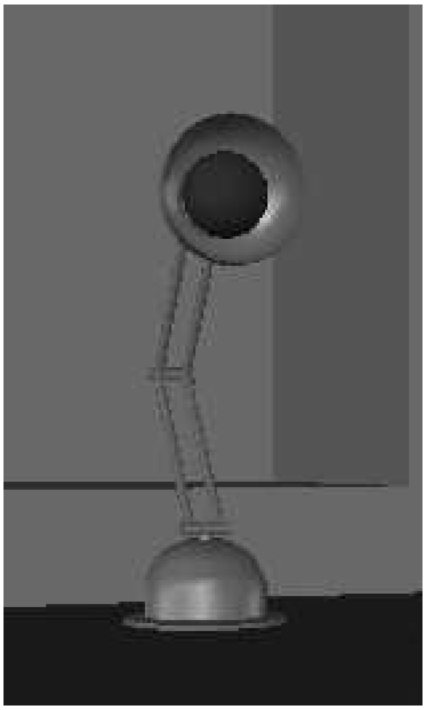

FIGURE 7.58: Headbanging movement 1

wishes of the agent's mind for what the agent will do, while the body fulfills these wishes as best it can given the constraints of the current situation, not all of which can be forethought by the designer or sensed rapidly enough by the agent. The motor system is needed because the run-time situation of the agent is uncertain; but it also means that *the action architecture* (and by extension the designer) *has no guarantee about the order or exact timing of the body's actions*.

But this exact timing is precisely what is at stake in generating expressive and clear animation. For example, when designing the Head-Banging behavior, I first used a keyframe editor to rough out the look of the behavior. Keyframe editors give direct, moment-by-moment control over the degrees of freedom of the body, immediately showing the effect of the chosen settings on the animation. Using the editor, it took about 5 minutes to generate a nice-looking animation.

The corresponding low-level signifier took days to implement. The behavior was not complex; the problem was not that the behavior would be incorrect. The problem was simply that you could not tell that the agent was purposefully hitting its head against the ground. Sometimes the lamp would look like it was flailing around; sometimes it would look like it was nodding; sometimes it would look like it was having a seizure; but rarely would it look like it was actually hitting its head on the floor. The difficulty is that head-banging involves multiple simultaneous actions: the agent must swing its head down while raising its body up (Figures 7.58-7.59), then swing its head up while bringing its body down (Figures 7.60-7.62), then snap its head back right before impact with the floor (Figures 7.63-7.64). All of these actions must be carefully timed,

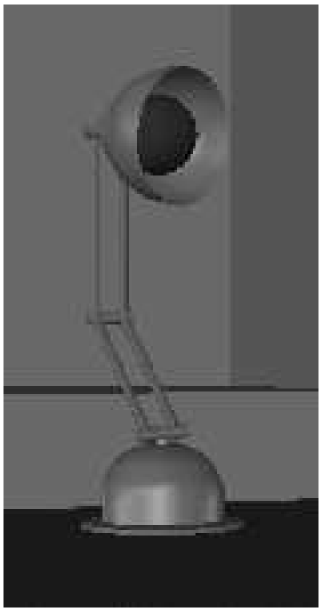

FIGURE 7.59: Headbanging movement 2

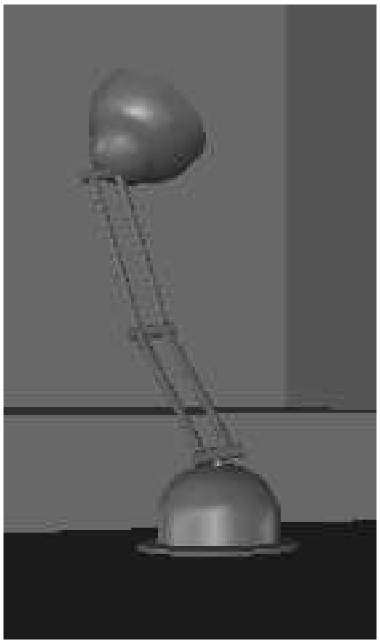

FIGURE 7.60: Headbanging movement 3

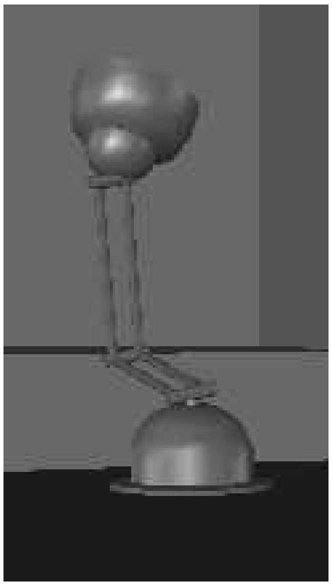

FIGURE 7.61: Headbanging movement 4

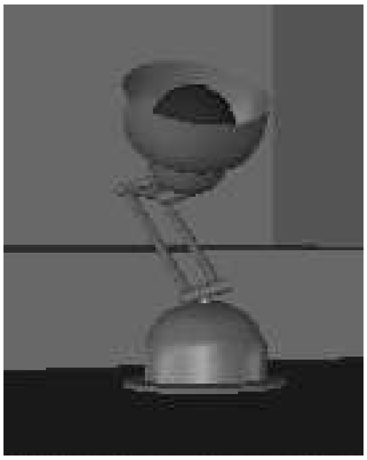

FIGURE 7.62: Headbanging movement 5

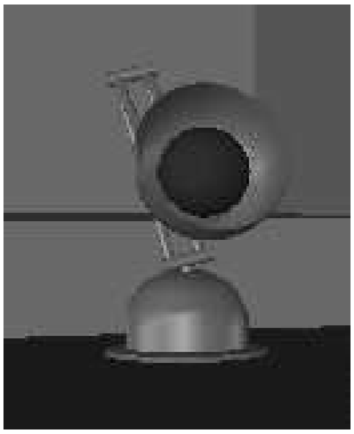

FIGURE 7.63: Headbanging movement 6

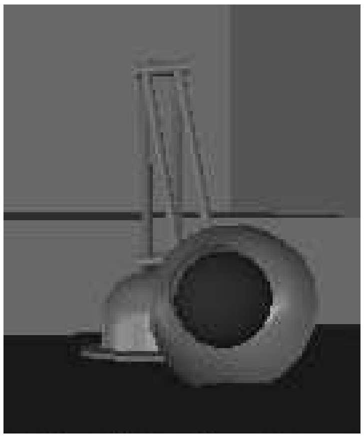

FIGURE 7.64: Headbanging movement 7

which is near to impossible in Hap; getting something that was remotely correct was a question of both luck and brute persistence.

The problem of generating expressive animation, while not a straightforward "AI problem," must be addressed by any architecture that is going to implement graphically presented, comprehensible agents. One promising avenue of exploration may be to use an automatic learning system such as genetic programming in order to generate code for the agent designer's desired low-level signifier. Automatic systems are easily able to generate many variations of behavior and test them rapidly in the virtual environment; these attributes could hopefully be harnessed to create the next generation of tools for expressive agents.

Expressivator: The Next Generation

If the problem of generating low-level signifiers is addressed, then the Expressivator suggests a new way of building agents. In the future, programming an agent might look like this:

1. Identify the agent's high-level signifiers.

2. Decompose the high-level signifiers into low-level signifiers.
3. Use machine learning to generate the low-level signifiers.
4. Identify mini-transitions between the low-level signifiers to make high-level signifiers.
5. Use machine learning to generate mini-transition sequences.
6. Write triggers for the mini-transitions.
7. Identify maxi-transitions between high-level signifiers.
8. Use machine learning to generate the maxi-transition sequences.
9. Write triggers for the maxi-transitions.
10. Tune everything by hand.

Transitions clearly add a new level of work for agent designers. Before, designers could content themselves to simply write behaviors. Now, designers must think about and implement many transitions between the behaviors. But in some sense transitions may actually *reduce* the complexity of the designer's job. Yes, you now need to write transitions, which was not necessary before; but transitions go between very simple behaviors with little internal structure, rather than the complex behaviors needed if one does not have transitions. And if you can generate most of the behavioral and transition sequences semi-automatically with machine learning techniques, in the end the behavior programming problem will be simplified.

Behavior Transition Types, Re-Visited

In Chapter 5, I argued for a range of transition types that the Expressivator should support. The Expressivator does, indeed, support all of the transition types I enumerated. Nevertheless, in practice I found that quite a few of the transition types were not useful. This is because the transition types are oriented towards blending or smoothing behaviors together. But for narratively expressive agents, the point is not to smooth behaviors but to make clear the relationships between them. Transition types that worked well to blur the distinction between behaviors worked poorly to explain the relationships between them; the reason for a behavioral change cannot be expressed when the user does not realize that the behaviors actually changed! Most of the mileage in transitions, then, comes from explanatory transitions; many of the other types were essentially clever tricks that do not help to make behavior more comprehensible.

Technical readers and curious humanists are now invited for a trip round section D.3 of the Appendix, which explains in more detail how each transition type was implemented, and whether or not it was useful.

Evaluation of the Expressivator

There are two aspects of the Expressivator that need to be evaluated:

1. *For designers*: Does the architecture give designers the controls that they need in order to implement the agents they may have in mind?

2. *For users*: Does the methodology behind the Expressivator actually create agents that are easier for users to understand?

Evaluation of the use of the Expressivator for the designer was part and parcel of the development of the Industrial Graveyard. In order to evaluate the effectiveness of the Expressivator in terms of what the user comes to understand, it would be best to do some kind of qualitative or quantitative user study. Unfortunately, this turned out to be beyond the scope of the thesis. In this section, I'll first explain the pluses and minuses of using the Expressivator to build an agent, and then discuss the ins and outs of how architectures like this one can be evaluated.

Evaluation for Designers

Advantages of the Architecture

One of the major goals of the system was to make it easier for designers to coordinate multiple high-level behaviors. This was successfully achieved. There is no doubt in my mind that behaviors are much easier to coordinate in the Expressivator than in Hap. This was underscored by my attempts to build the Overseer in Hap. Although the Overseer's activity is extremely simple, with clear conditions under which each behavior is appropriate, I spent many days trying to manipulate various Hap attributes to get each behavior to be engaged in at the right time. I finally gave up and let the Overseer use the behavior-killing meta-level control to delete old behaviors that were no longer relevant; without this hack it was simply impossible to control the Overseer well.

There are a number of problems with coordinating behavior in Hap that the Expressivator addresses:

- *The implicitness of behavioral choice*: In Hap, the choice of what behavior to pick at any time depends on a host of factors, including environmental conditions, priority differences between various behaviors and subbehaviors, and conflicts between behaviors. This means that getting a particular behavior to be chosen in a particular situation is a matter of manipulating multiple aspects of the agent design, not all of which have effects that can be straightforwardly understood. In the Expressivator, the designer writes triggers to cause behavioral change directly; having a behavior happen in particular circumstances means writing a single trigger that causes exactly that behavioral change.

- *The re-eruption of dormant behaviors*: Under Hap, when one behavior is chosen over another, the no longer chosen behavior remains in the agent's behavioral repertoire but becomes dormant.

Later, when the more important behavior is finished, the old behavior becomes active again. This works fine if the new behavior was a short interruption. But what also happens frequently is that the new behavior runs for quite some time; after it is done, the agent leaps back to an old behavior that has lost all relevance.

The Expressivator deals with this by actually deleting old behaviors when a new one takes over, instead of leaving them lying around to rear their forgotten heads later. I found that most of the time, behavior that has been interrupted for a long time should be started again from the top, instead of starting from whatever point the agent stopped at 5 minutes ago. The Expressivator makes this the default; in cases where the behavior should only be interrupted and not deleted, the special 'interruption' transition can be used instead.

- *Invisible behavioral interruption*: The problem of out-of-date behaviors suddenly becoming activated is compounded by the fact that in Hap, dormant behaviors, when re-awakened, do not actually know that they have been interrupted! Because they do not know they have been interrupted, they control the body as though there had been no lengthy break in their behavior, which is clearly wrong.

  For example, when building the Overseer I wanted the 'patrol' behavior to end automatically if it had been interrupted for quite some time; otherwise, the Overseer would try to return to whatever arbitrary point it had been walking to whenever other behaviors relinquished control of its body. Nothing worked properly except the extraordinarily simple measure of using the meta-level controls to kill the patrol behavior when you were doing something else more important. In the Expressivator, this problem vanishes because behaviors are deleted when they are interrupted; transitions explicitly inform behaviors when they are or are not active.

An additional major advantage of the Expressivator is the ability to clean up before and after behaviors. When switching from behavior to behavior, you have an opportunity to say something like, "I'm not reacting to the Overseer anymore, so I had better make sure to stop trembling and to squash a little less." For behaviors that have a large effect on body state — for example, that would involve the Patient tracking the Overseer, crouching down or stretching up, leaning over, keeping its eyes shut, or trembling — this opportunity to set the body back to a more appropriate state for the next behavior in some plausible manner is invaluable. Without it, the Patient has a good chance of repeating some of the major Woggle bugs: trembling or having its eyes shut through multiple subsequent behaviors, until some behavior serendipitously resets the body state.

But transitions do not seem essential to doing this clean-up activity. One possible way of doing this without transitions is to have a generic clean-up behavior, which you call before you start any behavior. I tried this with the Patient, but generally speaking this gave the look and feel of resetting the body after each behavior to a known state (the equivalent of Silas's standing up between behaviors mentioned in Chapter 5), which did not look good. Instead, I just made sure the transitionless version of the Patient avoided the most egregious behavioral carry-over by stopping trembling before every behavior. Nevertheless, there are still frequently problems in the transition-less version with inappropriate body aspects from previous behaviors carrying on into the next one.

A nice approach in general might be to define a clean-up behavior for each behavior. This clean-up behavior would reset aspects of the body state that the old behavior manipulates and that would probably be wrong in subsequent behavior. With the transition system, you know when a behavior is ending, so you could automatically call the clean-up behavior whenever the behavior was about to be deleted. This generic clean-up could occur in addition to whatever specific body changes were necessary for the next behavior.

Problems in the Architecture

There were certainly problems in the architecture. Of these, the most egregious is the problem of generating adequate animation. as discussed above. There were also some technical difficulties with the use of Hap as the basis language for the Expressivator, the most important of which is described in section D.4 of the Appendix for the benefit of technical readers; now would be a good time to take a look at it.

The major difficulty I ran into with the Expressivator per se (not its Hap substrate) is in reactivity. Specifically, in Hap, when you switch behaviors, the old behavior simply becomes dormant. The Expressivator, on the other hand, actually needs to delete the old behavior, including all its subbehaviors. This tended to add unwanted overhead to the time to switch — not much, perhaps 100 milliseconds, but enough to be noticeable in a delayed reaction time. One possible solution to this would be to simply mark behaviors as deleted, rather than actually deleting them; the agent could go back and actually do the work of deletion when it has more time to think.

Conceptually, though, the greatest problem with the Expressivator is the potential explosion of the number of transitions needed between signifiers. With 5 signifiers, there are up to 25 possible transitions; if an agent has 100 signifiers, there are far too many transitions to write by hand. From this perspective, the Patient has 24 signifiers, so it seems superficially like it would require just under 600 transitions!

But there are a number of factors, some theoretical, some practical, which cut down greatly on the number of actual transitions needed. An important factor in cutting down the number of transitions is the split between low- and high-level signifiers. Transitions are only needed between high-level signifiers, and between low-level signifiers that share the same high-level signifier — *not* between low-level signifiers in different high-level signifiers. This means that the Patient, with 8 high-level signifiers and 15 low-level signifiers grouped in small clusters, would require at most just under 150 transitions (64 maxi-transitions and 82 mini-transitions). In general, if we assume that low-level signifiers are distributed more or less evenly among high-level signifiers (rather than, say, all being under the same signifier), this reduces the original $O(n^2)$ problem to one of $O(n\sqrt{n})$).