4 An Agent-Based Approach to Animation

Introduction

In the last chapter, we saw how to build the model of the character's body, props and environment. We now move on to the question of **animation**, which in WavesWorld specifically refers to *the process of measuring and manipulating the model's articulated variables over time*. In a traditional computer graphics animation system, an animator would directly control the values of these variables over time, usually by setting their values at key frames or by attaching a spline to the variable and manipulating it directly. In a physically based computer animation system, the animator might simply allow simulated physics to run its course after setting up the initial conditions, and hope for the best.

In WavesWorld, however, the animator builds and configures a set of processes, called **agents**, that *measure* and *manipulate* the character's parts (i.e. the model built out of eve objects) over time. These agents become the eyes and hands of the animator. By building and layering configurations of agents together, the animator can build up complex characters that they and their fellow character constructionists can interact with at the task-level.

This chapter begins with a brief review of what "agent" means in WavesWorld. I then segue into some background material that should give some insight into this agent-based approach to animation. I briefly discuss my Master's thesis work, *Build-a-Dude* (**Johnson91**), and some of the issues that arose during and after that work. I then walk through a simple example of how one can use agents to build up the behavior generating components of a character with a minimal body (a point on a line).

I then give a more formal introduction to how agents are used in WavesWorld. I discuss some important issues that arise when building agents and hooking them together, with a special emphasis on **skill agents**, which are agents that can both **measure** and **manipulate** models. Finally, using some of the models I introduced in the

previous chapter, I explain a few more complex examples of how we can use agents to create a character's behavior, where **behavior** is loosely defined as the *result of processes (***agents***) measuring and manipulating a model's articulated variables over time*.

What's an Agent?

Before I begin discussing my agent-based approach to behavior, I need to briefly discuss what I mean by **agent**. Unfortunately, the term "agent" is a rather overloaded one, given both the amount of abuse it takes in the popular press and its dual nature in the AI literature. In this dissertation, I use the definition of agent proposed by Marvin Minsky in his book, *The Society of Mind*:

> *"Any part or process of the mind that by itself is simple enough to understand - even though the interactions among groups of such agents may produce phenomena that are much harder to understand."* (**Minsky8**6)

In WavesWorld, an agent really just refers to a software module; an autonomous black box that has a well defined purpose, a separate namespace and its own thread of control. I use the term very carefully, and whenever it is used in this dissertation, it explicitly refers to such a black box that, when opened, is readily understandable by someone who might build such things (i.e. a programmer).

Background

Build-a-Dude

In *Build-a-Dude* (**Johnson91**), the system I designed and built for my Master's thesis work, I eschewed dealing with the issue of a character's body, and looked instead at the composition of its behavior repetoire. I felt that if I allowed myself to either begin with the graphics, or to try and tackle both the mind and the body simultaneously, I would be both seduced and overwhelmed. Creating computer graphics is a compelling and heady experience, and my intent with Build-a-Dude was to concentrate on the less visceral, but still compelling, issue of the processes that generate a character's behavior. This system was notable in that it was implemented in a fault-tolerant, parallel, distributed fashion, where the processes generating behavior could run on a heterogeneous set of computing devices. Even so, there were severe restrictions that precluded easily adapting it for the task of building graphically simulated characters: the model it was manipulating over time

was really just text, and time was in discrete ticks, with all elements moving forward in lock-step.

I didn't realize it at the time, but there were really two problems I was dealing with in my SMVS work. The first was *what are the low-level processes that comprise the behavior repetoire of a character?* What are the building blocks; what are the essential elements?

The second question was *how are these elements configured?* What mechanisms do we use to sequence and select these elements' activity? At a base level, much of AI research has centered on these same issues, but I was interested in it with a very special slant towards building graphically simulated characters, not real robots or abstract software machines.

Planning Algorithms

One enormous help and impetus in this exploration was an algorithm (**Maes89**) developed by Pattie Maes, originally for robot locomotion. Situating my thinking with this algorithm was enormously useful, and it dovetailed nicely with work my advisor had been doing independently, trying to apply ideas from the ethological literature to the problem of computer animation (**Zeltzer83**, **Zeltzer91**, **Zeltzer88**). At the time, I tried to reshape Maes' action selection algorithm to fit all of the lower-level "motor planning" of an autonomous animated character. My implementation of this algorithm (and our extensions to it) are detailed in **Appendix A**.

Another problem that became apparent as I worked on Build-a-Dude and began working on WavesWorld, was the pervasive need for being able to debug what was being constructed. Given that what I was essentially trying to do was build a software development environment where the application under construction was to run in a distributed, parallel fashion over a wide range of resources, this should have been not been surprising.

Early Worlds

I then began trying to actually construct graphically simulated characters using an early version of WavesWorld. I built two worlds: raveWorld and sanderWorld.

raveWorld

RaveWorld was a collaborative one done with Abbe Don, and the intent was to experiment with multiple simple characters interacting with each other in a rich

environment. Each character had a torso and a head, where the torso had some texture map on it. The characters (**ravers**) were in a space where music was playing, where attributes of the music (song name and artist, beats per second, current lyric, etc.) were made "sense-able" to the characters. There were also three "video screens" in the environment, which were playing a series of images (by cycling through several thousand images from several sequences from films, texture-mapping them onto a polygon in the space). The ravers all had names, and favorite songs, colors, and video sequences. They also had other ravers that they liked and disliked, as well as songs they didn't like.

We built some simple skill agents, where a raver could dance to its favorite song, or move closer to a nearby raver it liked, move away from a raver it didn't like, texture map an image playing on one of the screens it liked, etc. We also built sensor agents to let it know when its favorite song was or wasn't playing, when a raver it liked or didn't like was nearby, etc. This all ran in real-time, with the display on a multi-processor SGI (running at ~11Hz), computation distributed some ten or so machines, and the sound and UI running on a NeXT machine.

sanderWorld

The next world I built was sanderWorld, which was inspired by the example used by Maes in her original paper on her spreading activation planner (**Maes89**). I built my robot sander as a simple 3D robot (actually a modified raver with two hands). It was in a simple environment running on the same multi-processor SGI that raveWorld ran on, where a user could manipulate the board with a data glove or by several other controls. I implemented a GUI to manipulate and visualize the parameters, running on a NeXT machine. Also, the agents in this world talked (using a variety of sampled voices), where the audio was spatialized (sensor agents and some skill agents were on the left, goal agents and the remaining skill agents were on the right). Whenever an agent was executing, it described what it was doing by sequencing the appropriate sound samples (again, running on a NeXT machine).

Lessons Learned

Building these two worlds was invaluable. I learned several things, all of them the hard way. The first was that the elements in the environment that are obviously perceivable to a user (or viewer) must also be perceivable to the character; if it's not, a user is immediately fixated on "why didn't character see that?" This included things that

aren't visual; i.e. "why is the character still dancing if the music stopped?"

The next thing I learned was how important *timing* was. At this point, I was only using an arbitary measure of time that was completely machine dependent to write behaviors. If I ran the same agent on a faster machine, the behavior executed faster. This was clearly unacceptable, but it was difficult to see how to get around it. Also involved with this issue was the question of two different agents manipulating the same character at the same time, especially when they were both manipulating the same degree of freedom. Race conditions ensued; which one prevailed? This too was a problem with no easy answer.

Another problem, also related to this issue of timing, was the fact that the smarter and more aware the characters became (i.e. the more sensor agents they had active, and the more often their receptors were running), the slower the system ran. Since (at the time), WavesWorld was an interactive simulation system, this was a serious drain on resources, since I needed to constantly be tweaking what was running to keep a balance of a useful/interesting character with a reasonable frame rate.

In building raveWorld with Abbe Don, I began to come face to face with some of the difficulties of collaborating with someone who was a peer in the creative process but who had a very different skill set than my own. I realized that I needed to be able to package up what I was doing so that she could reuse it, and that I needed to make tools so that she could do the same for me.

It also became clear that the current underlying graphics system I was using, which only allowed polygonal models with phong shading and texture maps, was completely inadequate for the kind of animaton I was interested in exploring. I needed a more expressive underlying representation, one that would allow me to use state-of-the-art commercial modeling software, as well as facilitate quick construction of custom animatable models directly in the system, and to freely intermix the two. At this point I revisited an earlier assessment I had made, namely that implementing a modeling system atop the RenderMan® Interface would be too difficult At this point, in spring of 1993, I began construction of the eve object framework, as described in Chapter 3.

Revisiting Planning

As time went on and I began grappling with the difficult issues of trying build graphically realized characters in WavesWorld, I realized that my original effort in trying to apply the planning work done during and after Build-a-Dude to a larger class of motor

problem solving was misguided. Maes' action selection algorithm, even extended for asynchronous action selection and parallel execution as we did in Build-a-Dude, was an excellent algorithm for a variety of motor planning activities, but it was no panacea. Many kinds of activities don't lend themselves to being cleanly articulated this way. How do you talk about reflex actions like blinking when a bright light passes by, or pulling your hand back before it gets burned from a sudden flame? It became clear that I needed facilities for building things in a more "messy" fashion, akin to the kind of accretive models Minsky proposes (**Minksy86**) . There are several problems that occur in trying to build a supportive infrastructure for messy behavior construction. Interestingly enough, many of these are analogues to the issues raised in trying to build a collaborative development environment. How do you let the left hand proceed without knowing what the right hand is doing, especially if the right hand is using the same resources, but in a way that is complementary, not conflicting? In other words, how do you facilitate behavior which blends together?

The Need for Building Blendable Behavior

For example, let's look at a character that has some set of autonomous skill agents inside it, each of which can measure and manipulate the character's state, where these skill agents have no knowledge of each other's existence. Furthermore, let's say we don't want to have to posit some sort of "manager" agent; we just want to have two independent agents acting on the same model, with no outside governor or modulating force. Let's look at the two following scenarios:

conflicting
You have two skill agents, **`pickUpTheCup`** and **`waveToFriend`**, both of which are executing. Among other things, both of them want to control the character's elbow and wrist rotation: **`pickUpTheCup`**, for example, wants to keep the elbow rotation at zero over the course of its activity, while **`waveToFriend`** wants to take it from zero to 45 and then back to zero. How should we resolve this? Does one win? Does neither? Do we negotiate a compromise? If so, how, and who initiates it—one side, the other, or a third party?

complementary
You have two skill agents, **`pickUpTheCup`** and **`beHungover`**, both of which are executing. Both of them want to control the character's elbow and wrist rotation: **`pickUpTheCup`**, for example, wants to keep the elbow rotation at zero over the course of its activity, while **`beHungover`** just wants to keep jittering it slightly. How should we resolve this? Does one win? Does neither? Do we negotiate a compromise? If so, how, and who initiates it—one side, the other, or a third party?

The easy answer here, especially in the first scenario, is some sort of resource locking

scheme: the first skill agent to say it needs something, gets it. The loser might continue to press its case for a bit, at which point the resource might be freed up, or it might just give up. Pretty simple to implement, certainly.

But consider the second scenario: if the **`beHungover`** skill agent is running for awhile, and then the **`pickUpTheCup`** agent gains control for a bit while it runs, at which point it ends and the **`beHungover`** skill agent takes over, that will look rather strange—the character was sitting there shaking slightly, suddenly he reaches out for a cup, and does it perfectly steadily, at which point he starts shaking again. Clearly, we'd like some mechanism for blending the results of the two skill agents; we'd like him to just seem to "reach for the cup in a hungover fashion." The skill agent picking up the cup doesn't necessarily know anything about the fact that the character is hungover right now.

Think about this from a collaborative construction perspective: we'd like to be able to mix and match agents from different authors together. Especially when we're just prototyping things, we don't want a given agent to have to take everything into account. After using these two agents for a while, we might realize that we might want to have a bit of shared state in the character that any of a variety of agents (say, any of the **`pickUp`*** or **`putDown`*** agents...) might use to perform their activity in a somewhat more or less hungover manner.

An Agent-Based Approach to Building Behavior

If we restrict an agent's activity to **measuring** and **manipulating** the **articulated variables** of a model built out of the eve object framework, as presented in the previous chapter, we get blendable behavior "for free". It might not be exactly what we want, and we may have to modify the character's agents and their interrelationships to get what we want. The point of WavesWorld is to provide the facilities for doing exactly this. If we want to build an agent which *does* act as a manager of other agents, it's straightforward: we could allow it to measure and manipulate the model as the other agents execute, and perhaps modify the relative weights of the two agents to allow one or the other to modify the model more or less.

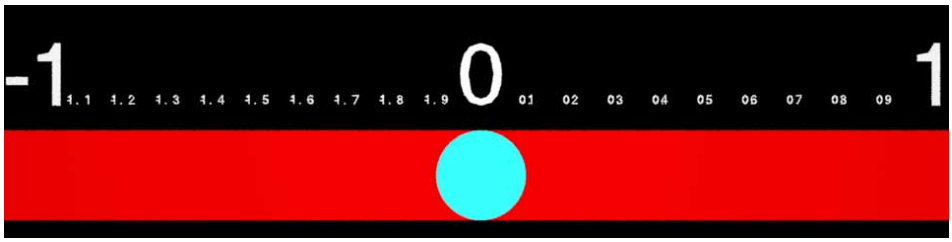
In the previous chapter, I presented a framework for constructing 3D models that have *the potential for change over time*. These models can be made out of reusable parts that can be composed together to build complex structures, which can be encapsulated ("packaged up") and reused in a collaborative construction environment.

This chapter presents an agent-based approach to building *reusable*, *composable*, *blendable* behavior. Used in the context of building three dimensional, semi-autonomous animated characters, it provides us with the power we need to create the "illusion of life" for our characters, bringing us closer to our final goal of being able to easily construct characters that can sense and act in the virtual environments we place them.

A Simple Example: Point's Point of View

In WavesWorld, I often consider the question of constructing a character from that character's perspective. From that vantage point, there is only the character itself (its behavior generating processes/agents and body/model)—everything else is considered the "world". In the last chapter we saw an approach for constructing a character's body (and its world), but what of its behavior? The parts of a character that generate its behavior have both drives (constant, long-running goals) and desires (once-only goals that disappear when satiated). These parts of the character must measure, perceive, and act on other parts of the character and the world. But what are these parts of the character, and how can they be perceived and acted on?

To facilitate the discussion, let us imagine the simplest interesting computational character we can: a point on line. The body of the character is a point, its mind is some amorphous collection of invisible processes and some shared blackboard, and its world is a line of some finite extent. To make this discussion a little clearer, let's refer to the character (the combination of its behavior generating processes and its body) as Point.



Let the virtual environment be completely defined as the line segment inclusively bounded by `xMin` and `xMax`, where `xMin < xMax`.

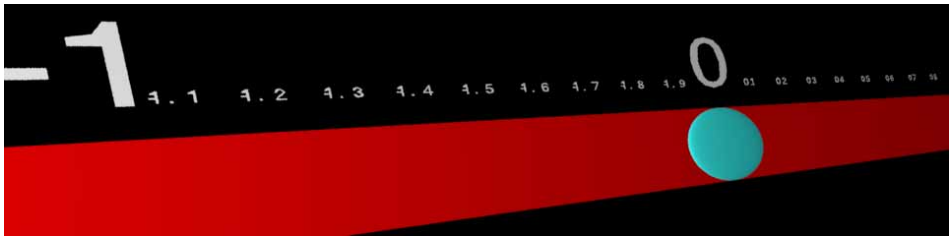
Point's body is completely defined by its location: `x = position(time)`

such that: `xMin <= x <= xMax`

and `position(time)` is a function defined somewhere in the character.

Point is rather single-minded: it just wants to stay at the location it considers "home", where home is a function also contained in Point that, when evaluated, returns the

desired value of x.



This implies several things: Point has some way of stating this desire (i.e. "I want to not be away from home"), it has some way of trying to fulfill this desire, and it has some way of measuring whether or not the desire is satisfied. Since each part of Point's mind is a simple but autonomous entity in its own right, we refer to each as an agent (see "What's an Agent", above).

We refer to the desire itself as a **goal agent**, the ability to affect the body and the world as a **skill agent**, and the perceptual mechanism that can sense whether or not the desire is satisfied as a **sensor agent**.

Skill Agent

They act in the virtual environment and/or on the body of the character. As opposed to sensory agents, skill agents can communicate directly with the virtual environment and the character's shape, shading, and state. e.g., **closeDoor**, **sitDown**

Sensor Agent

Connected to the virtual environment or the internal state of the character via its *receptors*, they only report back True or False. e.g., **doorIsOpen**, **IAmStanding**, **anEnemyIsNearby**

Goal Agent

They are defined in terms of sensor agents. This makes sense, because in order for a goal to be known to have been satisfied, the character needs to be able to *perceive* that it has been satisfied. e.g., **doorIsOpen**, **IAmStanding**

More precisely, in Point's case, its mind consists of:

- a sensor agent
- a goal agent
- a skill agent
- two function definitions:
- `point()`
- `home()`

The single sensor agent can be expressed as the function:

Note that the sensor agent is defined inversely from how you might think: it is defined

$$\text{sensorAgent}(\text{time}) = \begin{matrix} 0 & \text{if } (\text{position}(\text{time}) \equiv \text{home}(\text{time})) \\ 1 & \text{if } (\text{position}(\text{time}) \ncong \text{home}(\text{time})) \end{matrix}$$

as 1 (i.e. True) when Point is not at home, as opposed to True when he is at home.

The single goal agent can be expressed as the desired relation:

```
sensorAgent(time) != 1
```

It's especially important to note that the sensor agent and the goal agent use functions which may change over time, where there is assumed to be some single global time value that can be freely accessed. Since the character and its environment are distributed discrete systems, we also need to consider how often each of these functions are evaluated. There is some cost associated with evaluating any given function, so it is important to carefully consider how often it is evaluated, as the character has some finite set of computational and communication bandwidth resources.

By modulating the frequency at which the underlying functions are reevaluated, it is possible to embody the "attention" of the character to different facets of itself and its environment. For example, the sensor agent might only have evaluated the `home()` function at the beginning of a scene, at which time Point considered 12.7 to be home. Five minutes later however, Point, who may have been stranded far from home in the negative numbers, changes its mind and decides that it's content living at -77.2, and *that* is now considered home. This could happen if perhaps the goal agent, seemingly frustrated at the fact that it still has not been satisfied after some time, resigns itself to fate and thereby changes the definition of the `home()` function. If the sensor agent doesn't take the time to reevaluate the `home()` function, part of the society that comprises Point's mind will be under the delusion that it still longs for home, never realizing that other parts of its mind contain information that might avoid such consternation.

The solution I developed was inspired by a notion from Rasmussen who talked about the *signals*, *signs*, and *symbols* to which a virtual actor attends:

> "*Signals represent sensor data—e.g., heat, pressure, light—that can be processed as continuous variables. Signs are facts and features of the environment or the organism.*" *(***Rasmussen83**)

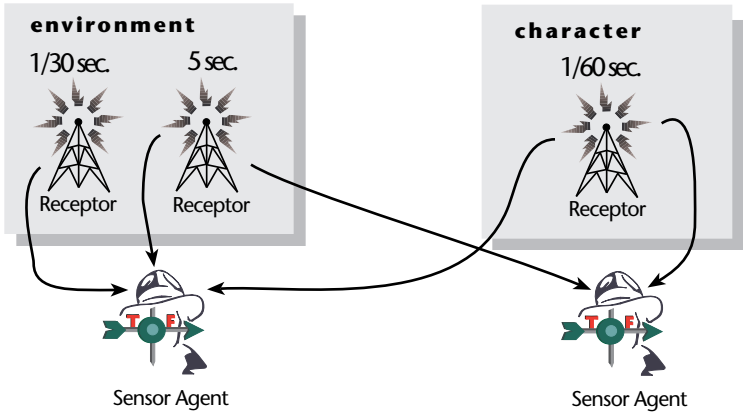
I realized that sensor agents corresponded directly to *signs*, but I needed some sort of representation for *signals*. In WavesWorld, these would be something that digitally sample continuous signals in either the character or its environment.

In addition to allowing a computational character to conserve its computational

resources, this is useful for encoding assumptions of the character about the dynamics of itself and its environment. Since agents are separate entities from the world, the character's body, and the other parts of the character's mind, they use computational devices called **receptors** to implement this distributed sampling.

Receptors consist of a probe and a transmitter that are "injected" by agents into either the virtual actor (i.e. the shared blackboard(s) among its behavior generating processes or in its body) or the virtual environment. Each receptor has a sampling frequency associated with it that can be modified by the agent that injected it. These probes sample at some rate (say, every 1/30 of a second or every 1/2 hour) and if their value changes by some epsilon from one sample to the next, they transmit a message containing the new value to the appropriate agents, which prompts the agent to recalculate itself.

Each receptor computes its value at a given sampling rate. If the value changes significantly from the last time, it sends a message to the sensor agents it is connected to.



One of the driving assumptions behind the receptors/sensor agents split is that a receptor will be evaluated many times relative to the sensor agent, so a given receptor should be computationally simpler than the sensor agents using them. They represent an assumption that communication is expensive compared to computation; it is cheaper for a sensor agent to embed a computation somewhere else than to keep asking for the value and determining itself if the value has changed enough for it to recalculate itself.

Another important function of the receptor/sensor agent split is to separate what can be directly measured in the character and the environment vs. what suppositions can be made using that information. In other words, a careful distinction is made from **signals** (i.e. what can be measured directly) and **signs** (what assertions can be made from what is measured). In Point's simple world, only its notion of where home is and where Point currently is can be measured-nothing more. But using just that information, we could

write a function called `howAmIFeeling()` that would allow Point to be `happy` (perhaps if it's close to home), `sad` (if it's far from home), `frightened` (if it's very far from home), etc. It's important to note that it may or may not have ways of expressing these states in its body, if the degrees of freedom in its mind are greater than the degrees of freedom in its body.

So what if Point has some desire which is unfulfilled? Assuming conditions are right, its skill agent is invoked. The skill agent's exterior is defined by:

- the conditions it needs to be true before it can be invoked
- a set of predictions about the world after it completes
- a black box which implements its functionality

Both the conditions and predictions are expressed with regard to sensor agents, since they must be perceivable by the character to be used.

In the case of Point's single skill agent, it requires that Point's single sensor agent returns 1, and it predicts that once it has completed, the sensor agent will return 0. Assuming it is called, it then executes the black box that implements its functionality.

This feedback loop—a goal agent expresses a desire, a sensor agent perceives that it is unfulfilled, and a skill agent is invoked in an attempt to satisfy it—forms the basis of a semi-autonomous character's behavior in a virtual environment. As more agents are added to the character, the interconnections of desire and perception become more complex, requiring more complex planning mechanisms. One such approach was described and implemented in my SMVS thesis system, and its current successor is used in WavesWorld—see **Appendix A**.

For the last several years, I've become more interested in simpler configurations of agents, since many kinds of interesting activity can be straightforwardly modeled as a very small collection of interconnected agents. For example, a single sensor agent gating a single skill agent, a sensor agent instantiating another small network of agents when true, or a sensor agent initiating a "reflex chain" (**Gallistel80**) of skill agents. From an animator's perspective, many times it can be simpler to break down the process of animating a character into separate parallel or sequential processes, where the animator has the freedom to keep separate or combine these processes as they see fit. Because the modeling approach I've developed allows me to have independent agents manipulating the same parts of a model without explicit knowledge of each other, this approach is now

possible. Because the models that can be built with this system are so rich, the activities of a single agent can be quite interesting.

For example,

- a character blinking when a bright light passes across its eyes
- a character standing up, walking to a door and opening it

can both be called "behaviors", even though the first is a relatively simple configuration of agents (perhaps a single agent to perceive the light and another to manipulate the eyelid when told to by the first) and the other is rather complex (some tens of perceptual agents measuring properties of the character and its immediate environment and several complex agents manipulating the character's body parts in concert).

Agents in WavesWorld

As I said near the beginning of this chapter, an agent in WavesWorld is an independent process with a well defined purpose; a computational black box that has its own namespace and a separate thread of control. In WavesWorld, **characters** are composed of a **model** (built using the eve object framework, as we saw in the last chapter) and some set of **agents**, interconnected with each other and the model in various ways. I realize that any discussion of agents will be confusing for some set of readers, because of the variety of the overloaded meanings people associate with the word. In WavesWorld, an agent is a “useful fiction” for thinking about an independent process’ relationship to a model over time. Agents don’t have to be implemented in a particular language or run on a particular kind of machine; they’re merely a way of thinking about any process which might be part of a character and be measuring and/or manipulating the character’s model over time.

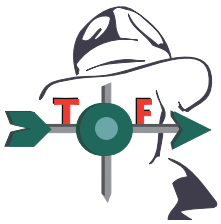
Agents allow the character to:

- have explicit goals
- perceive
  - the character
  - the environment
- act on
  - the character
  - the environment

In WavesWorld, there are three kinds of agents:

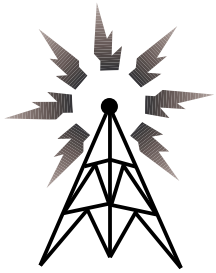
- Sensor Agents
- Goal Agents
- Skill Agents

Sensor Agents



Sensor agents allow the character to perceive itself and its environment at a level above the articulated variables of a model. These articulated variables are discretely sampled using a mechanism called a **receptor**. A sensor agent computes using the information gleaned from its receptors and renders a boolean assessment of the item it perceives (i.e. enemyIsNearby T, iAmSitting F, aCupIsNearby T, etc.)

Receptors



Receptors are not agents; they are used by agents (notably sensor agents) to discretely sample some articulated variable. In WavesWorld, receptors are a small piece of code that get embedded in some other process, where the host process evaluates the receptor at some given frequency. Receptors are comprised of the following:

- a signal name they measure (i.e. pressure, elbow position, mood)
- a sampling frequency (1/30 second, 30 seconds, 10 minutes)
- some epsilon function

Receptors sample their associated signal at their given frequency; if the value changes by greater than epsilon (computed by the epsilon function), they send a msg to their sensor agent. By setting the frequency of a receptor, we're encoding the following assumptions:

- the frequency of this signal (i.e. will it alias at this sampling frequency?)
- our minimum reaction time (i.e. if this changes by epsilon, how fast might we need to recompute ourselves?)

As we mentioned in the Point example, receptors represent an assumption that communication is expensive compared to computation; it is cheaper for a sensor agent to embed a computation somewhere else than to keep asking for the value and determining itself if the value has changed enough for it to recalculate itself. For all the systems that WavesWorld components have been ported to, this has been a very safe assumption.

Another purpose of the receptor/sensor agent split is to further abstract a given

sensor agent from the particular character or environment that its perceiving, thereby allowing a broader reuse of a given agent.

Goal Agents

Goal agents embody an explicit goal, and are described in terms of sensor agents. Given that character needs to be able to perceive that a given goal is satisfied, it makes little sense to proclaim it, unless it can be determined to be true or false.

Different kinds of goals, characterized by their lifespan:

- persistent, constant goal that is always extant in the character, i.e.
  - stay alive
  - be balanced
- more transitory, fleeting impulse, i.e.
  - not hungry
  - cup in hand

Skill Agents

A skill agent can

- perceive
  - the character
  - its environment
- act on
  - the character
  - its environment

In Build-a-Dude, skill agents were just one of three kinds of agents connected together in an action selection network, but in WavesWorld, they have assumed a much more central role. While both goal agents and sensor agents can **measure** the character and its environment, only skill agents can both measure and **manipulate** it.

The Anatomy of a Skill Agent

When a skill agent is first invoked, it needs communications' access to three objects/ namespaces:

- the character
- its environment
- the global scene clock

In WavesWorld, this means that a skill agent has access to three sets of variables: some of which map to the character, some of which map to the environment, and some of which map to the global clock (these are read-only).

As the skill agent begins, it **attaches** (see *Articulated Variables: A Model's Namespace* in the previous chapter) to the variables in the character and the environment that it will be manipulating. As the skill agent executes, time is moving forward for the character (and, consequently, the skill agent). The skill agent has some activity that it's trying to perform over some period of time, and it does this by **measuring** and **manipulating** the variables that it has access to.

When the skill agent finishes, it **detaches** (see *Articulated Variables: A Model's Namespace* in the previous chapter) from the variables in the character and the environment that it was manipulating.

Reflex, Oscillator, or Servo-Mechanism

When thinking about building a skill agent, I've found it useful to draw on a taxonomy proposed by Gallistel in (**Gallistel80**):

> *"There are a few kinds of elementary units of behavior. ...three of the most important kinds - the reflex, the oscillator, and the servomechanism. In a reflex, the signals that initiate contraction originate in sensory receptors. Although the reflex movement may affect the sensory receptors, one does not have to consider this "feedback" effect in formulating the principles that determine a reflex movement. In an oscillatory behavior, the muscle contractions repeat in rhytmic fashion. The contraction-triggering signals originate in a central neural oscillator, a neuron or neural circuit that puts out rhythmically patterned signals - a sort of neural metronome. In a servomechanism, the signals that initiate contraction originate from a discrepancy between two input signals. At least one of the input signals originates at a sensory receptor. The other input signal may originate at a receptor or it may be a signal of complex central organization. When there is a discrepancy between the two input signals, a third signal arises, called the error signal. The error signal initiates or controls muscular action. The resulting movements tend to reduce the discrepancy between the two inputs, thereby reducing the error signal. This is called negative feedback, for obvious reasons. In servomechanisms the precise character, amount, and timing of this negative feedback are a crucial aspect of the mechanism's functioning.* (**Gallistel80**, pp 10-11)

Writing a Skill Agent

The process of writing a skill agent is, as you would expect, an iterative one. The idea is to get together something quickly so you can begin editing. As with shaders (see *"Behavior" Shaders: A Useful Analogy* , later in this chapter), it's usually best to start with an already functioning skill agent. Skill agents tend to be a process which has some set of *reflexes*, *oscillators*, and *servomechanisms* which, depending on information it obtains by measuring the model, it invokes over the course of some amount of time to manipulate the model.

Many times the best way is to begin by constructing the skill agent is simply as a process which invokes single reflex, where the skill agent merely sets the articulated variables of the parts of the model it is concerned with. Because of communications delay in the system (just as in a living creature), an "instaneous reflex" never is, and this may actually lead to interesting behavior.

If, on the other hand, you don't want the articulated variables to change to some value immediately, but rather want to modulate them according to some pattern over time, you might consider building the skill agent as an oscillator. If so, you need to begin to deal with aliasing issues. In order to build a skill agent as an oscillator, you'll need at least the following pieces of information:

- the names of the initial state you'll be modifying
- the function over time you wish to apply to them
- the rate at which the world is being sampled

Note that depending on conventions for the model, you're working on, you may need more information. For example, if the model has all its variables normalized to some known range (i.e. 0 to 1), it's important to understand the dynamic range of those variables (i.e. for a color, is 0 black and 1 white, or a dingy grey?, etc.). If they report things in more absolute terms, you need to also get access to the extents. This is where conventions/stereotypes/prototypes/standards are vital in making the skill agent author's job easier.

Once you have the values you wish to manipulate over time and the range over which you want to change them, you then need to write your pattern generating function. This function will have at least one argument: time. You need to take care that the frequency of the signal being generated by this function does not exceed the frequency at which this skill agent is maximally sampled.

Using a copy of the model which is not being affected by other agents, run the skill agent see if it produces the desired animation. Test it with a variety of time lengths and sampling rates, to see how well it works in low sampling rate situations, and see over what range of time lengths it has the desired efect. Consider what other initial conditions might be interesting to use to modulate the setup for this reflex.

Many times the duration of a skill can itself become a parameter of the skill. For example, the act of opening and closing an eye, executed over a quarter of a second, is a

blink, but when it takes 3 seconds, it's most of a wink (a wink probably holds the close position longer than a blink). It important to understand the range of the parameters for which the skill is valid.

Finally, you might consider how to write the skill agent as a set of servomechanisms. These kind of skills are ones that are in a feedback loop with the variables they are manipulating in the model or environment. For example, imagine a simple reach skill for Rocky. Given an object name, he might find out where that object is and beginning rotating his arm towards the object over the course of some amount of time. Depending on how dynamic the environment is, we may see wildly different behavior depending on how often the skill agent rechecks the value of the object's location after he started. It may periodically check the value and attempt to correct the heading based on that.

On the other hand, we might have some set of servomechanism to choose from where the skill agent may have some notion of "patience". If the object didn't move, the skill agent would use a simple servomechanism to move the arm towards the object. If the error signal the servomechanism stayed low enough (i.e. the object wasn't moving away much), the character would seem to be following it and grab it eventually. If the object moved more, the skill agent might decide to "grab" for it, i.e. change the amount of time it was allowing the servomechanism to take to reduce the error signal, or it might "give up", i.e. just stop the servomechanism and exit.

The Importance of Conventions

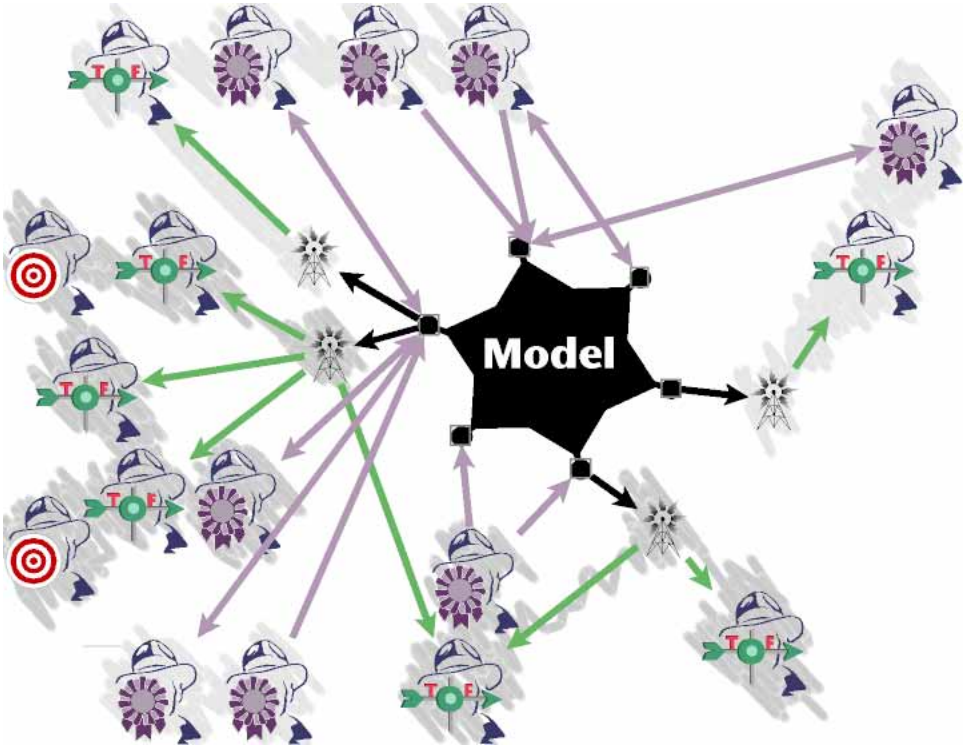
I can't overemphasize how important the conventions of a group, stereotype, genre, etc. are. Building a skill agent is really only a small (but important!) part of building a character, and it depends on a large number of factors a skill agent author may or may not have control of. How are variables named? Which ones can we assume are there? Are they normalized? What do the ranges mean? Are model parts set (i.e. all the characters have a "left" and a "right") or are they parameterized (it has "howMany" hands)? Again, the point of WavesWorld is to act as a testbed—for a given situation, any of these answers may be the correct one; we want to facilitate exploration of the options.

Each of the models we've built for WavesWorld can be thought of as *a set of possibilities*. Each model is really a *virtual actor*; able to play a variety of *roles*, becoming an infinite number of characters, depending on the particular initial values of their articulated variables and the agents that are attached to that model.

When building a skill agent, ideally, we want to always be thinking about how to write it for more general use. When we hard-wire something (a necessary sampling rate, a particular duration of the skill execution, etc.), we need to mark it as such. Ideally, we're never writing a skill agent from scratch, and we're never writing it for a particular model. Rather, we use a more prototype-based approach: we have a skill agent that "sort of" does what we want, and we have some "kind" of model we want to build it for.

Different Agent Configurations

There are many different configurations of agents that are useful to embody behavior.



We could use complex interconnections of agents like what I did in Build-a-Dude (see Appendix A and the discussion earlier in this chapter in *Early Worlds*), we could adapt newer algorithms like the one initially described by Blumberg in (**Blumberg94**) and implemented in By Blumberg and Galyean in (**Blumberg95**). We can also use simpler configurations like those proposed by ethologists (**Galistel80**, **Tinbergen51**) or "messier" ones proposed by AI researchers (**Minsky86**, **Travers90**).

Because of the ability of the underlying representation to support blendable behavior, we can freely intermix a variety of configurations of agents for a given character.

Trading Complexity: Model vs. Behavior/Space vs. Time

As we saw in Chapter 2, the problem of describing a character's behavior spans many fields, with many different approaches to the problem. All of these approaches, though, embody some set of trade-offs between the complexity of various parts of the character and the processes acting on them. There are a large number of factors that come into play here, for example:

- How general is the character?
- What environments will it find itself in?
- Will its parts be reused in other characters?
- Are we reusing parts from other characters?
- How generally does it need to be able to perform the tasks we've set before it?
- Is the environment dynamic?
- Is it hostile?
- Will the agents in the character be running on slow computers?
- Will the agents be distributed over a slow network?
- How much time does an agent have to react?
- Is the world moving forward too quickly (i.e. a real-time situation), or will it wait for a given agent to make up its mind once it's said it wants to act (i.e. a production situation)?
- Are all the parts working in concert or is the character a set of competing fiefdoms of agents?
- If it's the latter, do the competing groups of agents know about each other?
- Are we concerned with nuances of behavior or is gross activity the major concern?

The answers to all of the above questions can vary from situation to situation. All answers are, in a given context, potentially correct. This is what makes the character construction process such a difficult one to elucidate, as we need to discuss these tradeoffs explicitly. Even though a character designer may not discuss them, or even be consciously or unconsciously aware of them, they are making such tradeoffs based on their assumptions about the answers to questions like the above set.

For example, let's say we have a humanoid character that has two arms, like Rocky from Chapter 3. Using Rocky as a prototype, we might build an agent which moves the arm towards some object and picks it up. If the model is somewhat unconstrained (perhaps it only has min/max rotational constraints), this might entail setting the rotation parameters of the parts of the arm directly over time, i.e. forward kinematic control. On the other hand, the model actually might just have two parameters for the arm: a 3 space

location and a posture variable that varies from open to clenched. Either way, we could then write an agent to manipulate the various parameters of the model over time.

Now let's say that we want to use Rocky in the role of a private detective in a film noir, and we want to have the character be able to perform some of its activities in a "hungover" fashion (i.e. the role calls for a stereotypical, alchohol abusing, detective). The character designer has a choice: do they modify the model to have a parameter that corresponds to "being hungover", or do they change the agent to manipulate the model in a more or less "hungover" fashion?

So let's assume they attempt the former; how could we go about making a parameter of the model correspond to being "hungover"? Well, given a model parameter, we might apply some amount of stochastic jitter to it modulated by the "isHungover" parameter, which might vary from 0 to 1. We also might "roll off" the max of certain parameters, i.e. the shoulders can't be held as high as they had been, or the neck is always inclined. Also, purely visual parameters like how bloodshot his eyes are, or how constricted his pupils are, might also be affected.

On the other hand, we might modify the particular agent to take a parameter which does a similar modulation. Also, since the agent is working over time, it might decide to do the activity more slowly, or less accurately, depending on that parameter value.

The point here is not that one or the other approach is more correct. We could see how there are advantages and disadvantages to both, each implying different tradeoffs. If we modify the model, all of our behaviors can potentially take advantage of this new parameter. On the other hand, there may be time-based changes to the model (having something happen more or less slowly) that are difficult to do in the model but are straightforward to do in the agent. On the other hand, by putting the information and manipulation in the agent, we can solve the particular problem at hand in a well controlled and understood fashion without having to worry with the issues involved in generalizing it for other behaviors (standing up, walking, etc.).

Also, one of the aims of WavesWorld is building a collaborative environment: in other words, the model designer might not be (and probably isn't) the behavior designer, and therefore it might be non-trivial for the person building the agent to make the necessary modifications to the model.

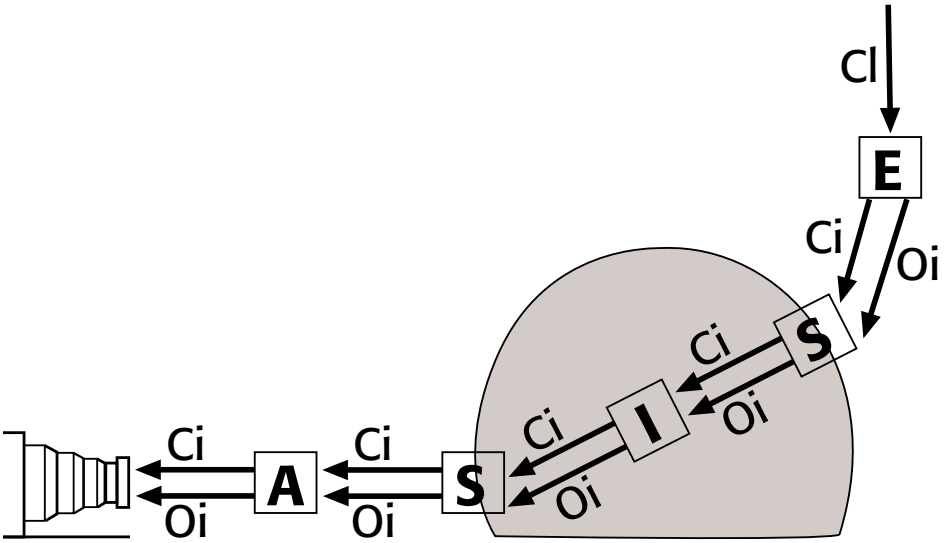
Ideally, we would be supportive of both activities, where perhaps a single agent is

modified to do something in a particular fashion. After using this agent and understanding what might be entailed, this activity could be slightly generalized to a few other similar agents. After using them, it might become clear how this kind of activity could be easily supported by an extension of the model. The animator could then contact the modeler and ask for this extension to be built in. In this way, building behaviors could be a way of prototyping the kind of malleability one might need from a given class of models. This sort of iterative, collaborative design is exactly what I'm trying to foment with WavesWorld.

"Behavior" Shaders: A Useful Analogy

In order to facilitate our discussion of building behaviors, I've found it useful to draw an analogy from an area intimately related to computer graphics: the notion of using black box *shaders* in a renderer.

Several Computer Graphics systems (RenderMan®, Apple's QuickDraw3D) allow constructing custom boxes that get called by the renderer during various points in rendering pipeline. For example, in RenderMan®, the shading pipeline looks like this:



A colored ray of light (Cl) starts from the upper right and moves through the exterior atmosphere (E) to strike the surface (S). It moves through the interior volume (I) of the object and comes out the other side (S) where it travels through the atmosphere (A) to finally strike the camera's recording plate.

The details of this diagram aren't relevant here, but the point is that the boxes (containing "A", "S", "I", "S", "E", from left to right) are called **shaders**, and they can be easily replaced and modified, giving precise control over all important aspects of

the rendering process. RenderMan® shaders are written in a special purpose language that looks alot like C. Like C code, shaders must be compiled before they can be used. Shaders have parameters, and the user can specify if this parameters are fixed on the model or if they change over the surface of the model.

Interestingly enough, looking at the process of designing and writing shaders has resonances in WavesWorld for both the process of designing models (which like shaders, are only visible by their parameters, which can change over time) and to building skill agents (which generate behaviors discretely sampled signals, albeit behaviors are time-based and shaders are in parameter space (xyz, uv, st)).

RenderMan®'s Shading Language is a compiled language that has no standard virtual machine model that it compiles to, so in order to share shaders among different RenderMan® Interface compliant renderers, the source code must be available. Within a given collaborative space (i.e. a single production company or research group), this may be feasible, but commercial shader writers rarely, if ever, make their source code available. Because you can't blend most shaders together easily (light source shaders being the one exception), this makes it difficult to reuse shaders for which you don't have source code. Also, because shaders tend to be written as single monolithic functions, it's difficult to package up shaders in anything other than the most obvious ways (i.e. default values).

Because shaders have no control over how often they are called (the renderer deals with this), a shader writer has to understand and deal with antialiasing issues themselves. This implies some set of assumptions and tradeoffs the shader writer is willing to make, and without appropriately built models (uv, attention to scale, etc.), the shader's job is more difficult and limited in terms of what it's able to do.

It's interesting to think about how "behavior" shaders differ from RenderMan® shaders:

- extent
  - RenderMan® shaders concern themselves with sampling in space (i.e. u v).
  - "Behavior" shaders concern themselves with sampling over time.

- execution cost
  - RenderMan® shaders are called many (thousands or millions) times a frame.
  - "Behavior" shaders will be called at most once per simulation time sample; of which there probably won't be more than 100 per second.

- sampled data
  - RenderMan® shaders are not all procedural; many use texture maps for bump, environment, shadow, and a myriad of other mappings besides the obvious of painting an image on a surface.
  - “Behavior” shaders can use motion capture information to the same effect, using it to guide or shape other procedural (or other sampled data) data.

One-Offs/Special Purpose

In Pixar's short film "KnickKnack", a lovesick snowman tries vainly to escape from his glass snowglobe. At one point, he attempts to blow up the walls, but only manages to start a whirling snowstorm and scorch his carrot nose.

image © Pixar, used with permission

For this scene, which doesn't last more than ten seconds or so, a special purpose shader was written to do the shading of the burnt carrot nose. That shader was custom written for that purpose; and was actually the first shader written by that technical director (**Miló93**). The effect of this shader is not overwhelming or central to the story, but then again, the TD who wrote it didn't labor over it that long. The fact that this shader was very, very special purpose and only used once is fine, because it didn't require much effort to write. This kind of curve is important to consider; we want to be able to accomplish something quickly, and also have enough headroom so that if the behavior designer wants to put more effort in, it's also worth their while.

Reusable/General Purpose

On the other hand, there are certain shaders that are used again and again, in many different situations and for many different reasons. One such ubiquitous shader is Larry

Gritz's wood shader, an example result of which can be seen here:

This shader is used again and again in environments that I and hundreds of others who use Larry's software around the world build. It has parameters for changing the size of the planks, the kind of wood, the color of the wood, the size and color of the grooves, etc. This shader antialiases itself very well; it looks great even at one sample per pixel. Larry spent a long time writing this shader, and this is reflected in the flexibility and robustness of the shader. Both of these paradigms, building both one-offs and general, reusable modules, need to be supported in a testbed for constructing 3D semi-autonomous characters. The goal is not always a highly tuned, general, reusable module, not when the time spent is not warranted, but there is a need for "something". We can learn valuable lessons from our compatriot shader writers.

In the following examples, I'll be discussing how we animate the character parts and props we saw how to build in the last chapter. In contrast to last chapter, where I introduced a modeling language (**eve**), there is no special language in WavesWorld for behavior; just an approach. One thing that may be confusing are the code fragments that I use to illustrate the agents that are generating behaviors. As I discussed earlier in this chapter, the term agent is a "useful fiction", used to refer to any independent process that measures and manipulates a character's model. They can be written in an language, and can take a variety of forms. For consistency's sake, and because most of my agents are written use my tcl-based active object tools, I've used tcl as the language to illustrate these examples, but they could just as easily have been written in any compiled or interpreted language which has been supplemented with a communication library to allow it to communicate with the eve run-time system.

The command **`defineMotorSkill:`** defines a new "motor skill", which is a routine an agent calls to measure and manipulate a model's (either the character's or the environment's) variables. It is assumed that the agent has attached to the eve run-time system, and then invokes the motor skill. After the motor skill exits, the agent flushes its communications channel to the run-time system.

In addition to defining motor skills and the extensions discussed in the the last chapter (see *Tcl Additions*), the flavor of tcl used here has several other routines which may look unfamiliar to readers:

fromCharacter: *varName1 varName2 ... varNameN*
This command marks the variables *varName1 varName2 ... varNameN* as variables that are external to this agent and located in the character; i.e. these are articulated variables in the model. Reading from and writing to one of these variables involves some communications cost. Writes are potentially cached until the next invocation of **synchWithScene**.

fromEnvironment: *varName1 varName2 ... varNameN*
This command marks the variables *varName1 varName2 ... varNameN* as variables that are external to this agent and located outside of the character; i.e. these are articulated variables not associated with the character's model. Reading from and writing to one of these variables involves some communications cost. Writes are potentially cached until the next invocation of **synchWithScene**.

synchWithScene
This command flushes writes to the eve run-time system from the agent. It returns the amount of time that has passed since the last time this agent did a synchWithScene, in units of "ticks", where a tick corresponds to the smallest unit of time increment for the global scene clock (i.e. `1./$scene(ticksPerSecond)`). Note that a motor skill has this happen automatically immediately before it is called,

and immediately after.

Also, the variable `scene` is available, which contains several useful pieces information, including the current global scene time, the current maximum sampling rate of time (i.e. how fast or slow is the global scene time incrementing), etc.

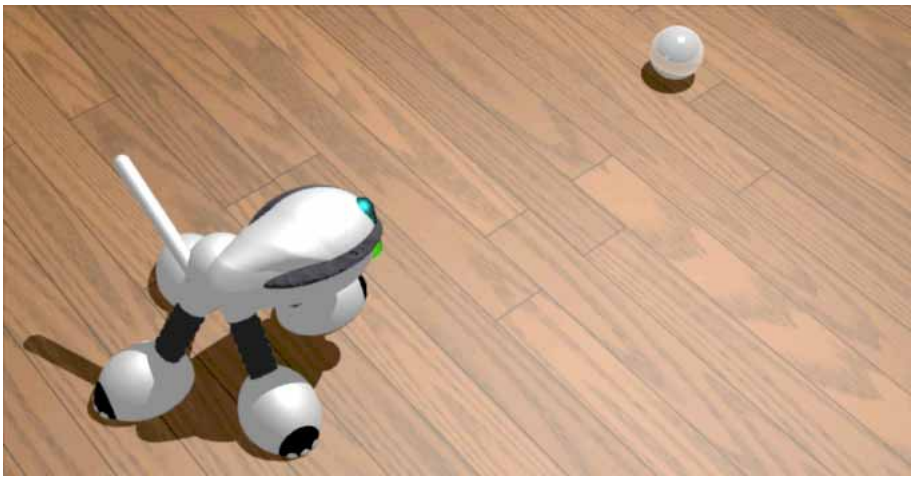
A Simple Orient Behavior

A rather ubiquitous behavior is that of "orienting"; given some object, the character should focus its attention on it. In this example, I'll discuss the development of a simple orient behavior for robbit. The scenario is that there is a "shiny thing" in robbit's environment, and it has a single sensor agent which tells it that its nearby.

We tell robbit to orient towards the object, and, if the sensor agent is true, the skill executes. The task is to think about how we might implement that skill agent. With robbit, we have many immediately useful degrees of freedom in the model:

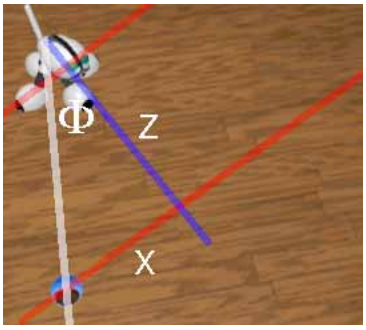
- he can move his glowing "eye" towards the point.
- he can turn his head towards it
- he can look up
- he can rotate his body to orient towards the object
- he can can twist his to look at on the side; somewhat askew or quizzically

And of course he can do any combination of these, over time, with some given function (i.e. overshoot it, ease in, whip in, etc.). As an initial experiment, I wrote a little procedure to have robbit orient itself toward the shiny thing by rotating about in Y:



```
defineMotorSkill: orient {} {

   fromEnvironment: shinyThing
   fromCharacter: robbit

   set xDiff [expr {$shinyThing(xTranslate) - $robbit(xLocation)}]
   set zDiff [expr {$shinyThing(zTranslate) - $robbit(zLocation)}]

   set yR [expr {degrees(atan([expr {$xDiff/$zDiff}]))}]

   set robbit(orientation) $yR
}
```

This was pretty straightforward; do a little trigonometry between the two objects:



and it's done. This is a simple example of thinking about how you might implement a skill as a servomechanism that works "instantaneously", where it computes the error between what it has and what it wants, and immediately increments the value to what it wants.

We could obviously do the same for his head, instead of his body. But now we might want to consider what the degrees of freedom here that we want to control and how we want to manipulate them over time. One idea might be that as Robbit turns, start with the smallest parts (like the eyes), then bring the other parts in line (like the head), and then finally turn the whole body and line it up, straightening out the other bits along the way. In

a sense, this is just a straightforward application of the principles of traditional animation: *anticipation*, *follow through* and *overlapping action* (**Thomas81**, **Lasseter87**).

In the case of Robbit, that would mean we might orient his glow to the side he will be turning, then turning his head some amount (stopping short if it's farther than, say, 45 degrees off center), and then twisting the body while reorienting the head. Let's consider some slight variations. What if Robbit briefly moved his head toward the shiny thing, then quickly back to center, then paused for a fair bit, then quickly rotated his head and followed with his body almost immediately. This would be very different than if he leisurely rotated his head, and then smoothly rotated his body at the same pace. So how might we write a routine that could at least initially do the latter? Well, here's one:

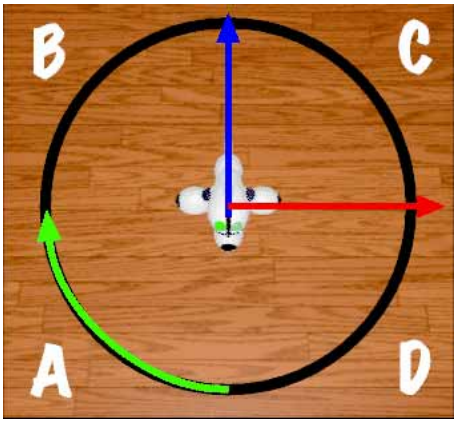
```
defineMotorSkill: orientBoth {{headTurnTime 1} {finalTime 2}} {
   fromCharacter: robbit head
   fromEnvironment: shinyThing

   set xDiff [expr {$shinyThing(xTranslate) - $robbit(xLocation)}]
   set zDiff [expr {$shinyThing(zTranslate) - $robbit(zLocation)}]

   set yR [expr {degrees(atan([expr {$xDiff/$zDiff}]))}]

   set amt [expr {$yR + (-1 * $robbit(orientation))}]

   set initialHead $head(leftRightR)

   if {$amt > $head(leftRightRMax)} \
   {  set max $head(leftRightRMax);
      set amtLeft [expr {$amt - $head(leftRightRMax)}];
   } \
   { set max $amt;
     set amtLeft 0;
   }

   # first move the head
   set steps [expr {$headTurnTime * $scene(ticksPerSecond)}]

   set uIncr  [expr 1.0/[expr {$steps - 1}]]
   set u 0
   while {$u <= 1.0} \
   {  set head(leftRightR) [expr lerpUp($u, $initialHead, $max)];
      set u [expr {$u + [expr {$uIncr * [synchWithScene]}]}]
   }

   # okay, at this point, we've moved the head either as far as it
   # will go or as far as it needs to go.  We now want to bring the body
   # around, unrolling the head (i.e. keeping it in the same absolute
   # position it's in now) the whole time.
   set steps [expr {$finalTime * $scene(ticksPerSecond)}]
   set uIncr  [expr 1.0/[expr {$steps - 1}]]
   set u 0
   set initialRobbit $robbit(orientation)

   while {$u <= 1.0} \
   {  set head(leftRightR) [expr lerpDown($u, $initialHead, $max)];
      set robbit(orientation) [expr lerpUp($u, $initialRobbit, $yR)];
      set u [expr {$u + [expr {$uIncr * [synchWithScene]}]}]
   }
}
```

So we might make a little movie of Robbit using that routine (I did; unfortunately it's impossible to make it available in the paper version of the disseration). The first time he takes one second to turn his head, two seconds to turn his body, then half a second to turn his head and one second to turn his body, and finally one second to turn his head, and and half a second to turn his body.

We might note that the head turn and body turn are really divorced from each other; the head feels like it's floating in space as the legs start coming around. We could overlap the action a bit. Maybe most of the way through the head turn, the body begins to turn, and then have the head bob a bit between its final point. Also, we might turn the toes a bit in the direction we're going, and straighten them out as we turn. We might also want to bob the head a bit.

If Robbit is at the origin and the shiny thing is (0, 0, -1), this works fine. But what if it the "shiny thing" is somewhere else, like say (1, 1, 3)? Well, that simple bit of trigonometry I did before breaks down. Let's think about the problem (when it goes out of

quadrant A):



Well, we might decide to do the following (*note to C programmers: I realize atan2() would solve this, but this is for demonstration purposes, so bear with me*):

```
defineMotorSkill: orient {} {
   fromCharacter: robbit head
   fromEnvironment: shinyThing


   set xDiff [expr {$shinyThing(xTranslate) - $robbit(xLocation)}]
   set zDiff [expr {$shinyThing(zTranslate) - $robbit(zLocation)}]

   #don't want a division by zero, do we?;
if {$zDiff == 0.0} {  set zDiff 0.001; } {};

   # As robbit(orientation) goes from 0 to 360, it moves
   #  from quadrant A, to B, to C, and finally D
   # If xDiff is negative, it's in quadrant A or B
   # If zDiff is negative, it's in quadrant A or D

   if {$xDiff < 0.0} \
   {  if {$zDiff < 0.0} \
      {  # it's in quadrant A;
         set robbit(orientation) [expr {degrees(atan([expr {$xDiff/
            $zDiff}]))}]
      } \
      {  # it's in quadrant B;
         set robbit(orientation) [expr {180 + degrees(atan([expr {$xDiff/
            $zDiff}]))}]
      };
   } \
   {  if {$zDiff < 0.0} \
      {  # it's in quadrant D;
         set robbit(orientation) [expr {360 + degrees(atan([expr {$xDiff/
            $zDiff}]))}]
      } \
      {  # it's in quadrant C;
         set robbit(orientation) [expr {180 + degrees(atan([expr {$xDiff/
            $zDiff}]))}]
      };
   }
}
```

Now if we look at the simple skill agent we wrote as a servomechanism above that could be used to orient both the head and the Robbit's body, we note that it was based on the same faulty assumption that the first **`orient`** motor skill was. So we might rewrite the **`orientBoth`** motorSkill to use this, but think about this: it always rotates clock-wise.

What if Robbit is oriented pointing at (0, x, -1) and we tell it to orient towards (1, 0, -1)? Well, with the naive method we've been using, it will spin all the way

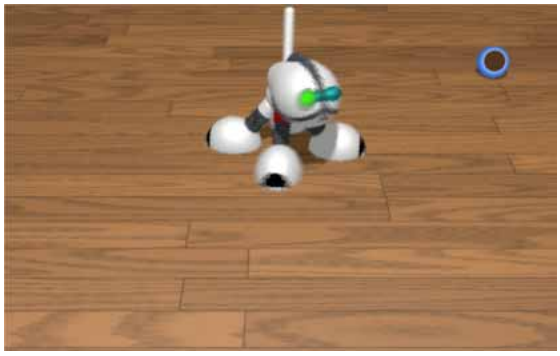
around to get to that point. Now that actually might be entertaining for a dog-like character (chasing his tail and all), but let's say we're looking for slightly more intelligent behavior. So how might we achieve it? If the amount of rotation we decide on is greater than 180, we just rotate `(-1 * (360 - $amt))`. In other words:

```
defineMotorSkill: orientBoth {{headTurnTime 1} {finalTime 2}} {
   fromCharacter: robbit head
   fromEnvironment: shinyThing


   set xDiff [expr {$shinyThing(xTranslate) - $robbit(xLocation)}]
   set zDiff [expr {$shinyThing(zTranslate) - $robbit(zLocation)}]

   if {$xDiff < 0.0} \
   {  if {$zDiff < 0.0} \
      {  # it's in quadrant A;
         set yR [expr {degrees(atan([expr {$xDiff/$zDiff}]))}]
      } \
      {  # it's in quadrant B;
         set yR [expr {180 + degrees(atan([expr {$xDiff/$zDiff}]))}]
      };
   } \
   {  if {$zDiff < 0.0} \
      {  # it's in quadrant D;
         set yR [expr {360 + degrees(atan([expr {$xDiff/$zDiff}]))}]
      } \
      {  # it's in quadrant C;
         set yR [expr {180 + degrees(atan([expr {$xDiff/$zDiff}]))}]
      };
   }

   # what's the shortest way to get there?
   if {$yR > 180} { set yR [expr {-1 * (360 - $yR)}]} {}

   set amt [expr {$yR + (-1 * $robbit(orientation))}]
   set initialHead $head(leftRightR)

   if {$amt > $head(leftRightRMax)} \
   {  set max $head(leftRightRMax);
      set amtLeft [expr {$amt - $head(leftRightRMax)}];
   } \
   { set max $amt;
     set amtLeft 0;
   }

   # first move the head
   set steps [expr {$headTurnTime * $scene(ticksPerSecond)}]
   set uIncr  [expr 1.0/[expr {$steps - 1}]]
   set u 0
   while {$u <= 1.0} \
   {  set head(leftRightR) [expr lerpUp($u, $initialHead, $max)];
      set u [expr {$u + [expr {$uIncr * [synchWithScene]}]}]
   }

   # okay, at this point, we've moved the head either as far as it
   # will go or as far as it needs to go.  We now want to bring the body
   # around, unrolling the head (i.e. keeping it in the same absolute
   # position it's in now) the whole time.
   set steps [expr {$finalTime * $scene(ticksPerSecond)}]
   set uIncr  [expr 1.0/[expr {$steps - 1}]]
   set u 0
   set initialRobbit $robbit(orientation)

   while {$u <= 1.0} \
   {  set head(leftRightR) [expr lerpDown($u, $initialHead, $max)];
      set robbit(orientation) [expr lerpUp($u, $initialRobbit, $yR)];
      set u [expr {$u + [expr {$uIncr * [synchWithScene]}]}]
   }
}
```

Can you recognize the assumptions built into the above routine? Look at at this way: what might change over the course of the routine (i.e. the code inside the while loops)

that we're not rechecking as we go? Well, the most obvious built-in assumption is the fact that **`synchWithScene`** returns a tick value, and the amount that we increment u is based on the assumption that `$scene(ticksPerSecond)` is constant over the lifespan of the routine. Is this a valid assumption? The answer, of course, depends on the particular virtual environment we drop Robbit into.

Applying this behavior and playing with Robbit in a variety of initial conditions, we could begin to understand how well this behavior works. Here's a shot during one these interactions with him as he's turning toward the "shiny thing":



At one point, as he spins, he goes twisting around, putting his head right through his tail. This probably isn't what we want. So what's wrong with this agent?

Let's think about what is different about the time he spins around and goes through his tail while orienting himself. Well, in this version of the motor skill, we now allow Robbit to go backwards; could that have an effect? Remember that while he can spin his body around 360 degrees, we constrain his head rotation. Right now, we haven't really constrained it in the model; we're just not setting it higher in the agent. So are we asking it to rotate around more than its max or less than its min? Sure we are—that's the bug. We constrain the max, but not the min. Therefore, we need to change:

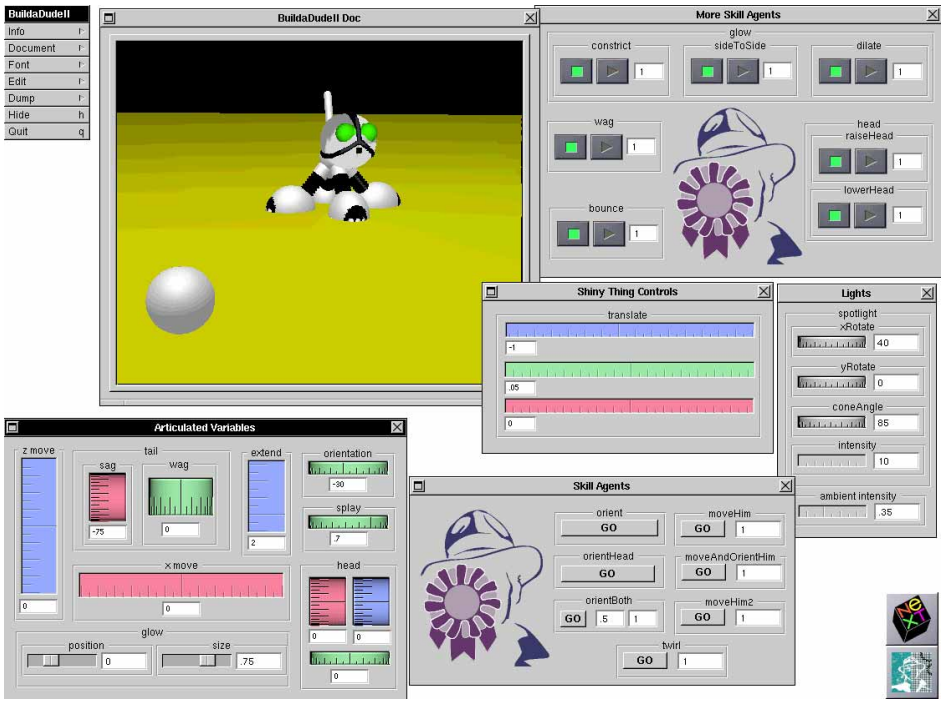
```
if {$amt >$head(leftRightRMax)} \
{  set max $head(leftRightRMax);
   set amtLeft [expr {$amt - $head(leftRightRMax)}];
} \
{ set max $amt;
  set amtLeft 0;
}
```

to be:

```
if {$amt >$head(leftRightRMax)} \
{  set max $head(leftRightRMax);
   set amtLeft [expr {$amt - $head(leftRightRMax)}];
} \
{  if {$amt <$head(leftRightRMin)} \
   {  set max $head(leftRightRMin);
      set amtLeft [expr {$amt - $head(leftRightRMin)}];
   } \
   { set max $amt;
     set amtLeft 0;
```

```
    }
}
```

So now we've got a pretty good skill agent for orienting robbit, which might be reusable for other models, but we didn't really consider that in building it. We'll think about building a reusable skill agent in the next section. We could obviously now add some more business (move the glowing eye around, we could play with twisting his feet, overshooting the head, etc.). Here's a screen shot from what the specific GUI I built for testing and experimenting with this:



At this point, if you play with robbit for a while, you might notice a "bug" in robbit's behavior. At one point, where you would think that robbit would turn to his right to look at the ball, he spins counter-clockwise (looks like 260 degrees or so) to look at the ball. So why is this? Maybe robbit is checking based on distance from 0 to the turn, not based on his current location?

Here was the problem. When I was checking to see if I need to approach the shiny thing clockwise or counter clockwise, I wasn't offsetting how much I had to turn by how far I'd already turned. In other words, this code:

```
if {$yR > 180} { set yR [expr {-1 * (360 - $yRyR)}]} {}
```

needed to become:

```
if {[expr {$yR - $robbit(orientation)}] > 180} { set yR [expr {-1 * (360
        - $yR)}]} {}
```

There are good reasons to leave the first behavior, though, especially if the character you're building isn't supposed to be too bright or is just somewhat contrary. We can add a **`robbit(contrary)`** variable which can be set true or false. If the robbit is feeling contrary, he'll go the long way. Otherwise, he'll go the shorter distance. So here's what that code looks like:

```
# if robbit is "contrary", he'll go the long way,
# otherwise go the short way
 if {$robbit(contrary)} \
 {  if {[expr {$yR - $robbit(orientation)}] < 180} \
    { set yR [expr {360 - $yR}]} {};
 } \
 {  if {[expr {$yR - $robbit(orientation)}] > 180} \
    { set yR [expr {-1 * (360 - $yR)}]} {};
 }
```

What we've seen

This section has walked through the process of building a non-trivial skill agent that can be used to generate useful and interesting behavior. The intent here was to give you the flavor of what process a non-expert skill agent builder might go through to construct such a skill agent. We really paid no attention to issues of designing for *reuse*, though, which is what we'll now turn our attention to in the next section.

Building a Reusable Behavior

Now that we've seen how we might construct a behavior by building a skill agent and hooking it to a sensor agent, let's take a look at AudreyeII and make her look left and then right. Recall that this model has a single variable in it that controls the left/right orientation of the eye. Basically, we'd like to build a behavior which, over some period of time, manipulates this variable to go from its initial value to some minimum value, then makes it go up to some maximum value, and finally brings it back down to the initial value. It's assumed that the initial value would lie somewhere between (or equal to) the minimum and maximum values. So what are some of the issues here?

The first has to do with how we view this activity occurring over time — are there three distinct phases, each with a predetermined portion of time allotted, or is this one continuous amount of distance to travel, at the same rate over all of it?

I point these issues out not because they're terribly important in this particular simple behavior's case, but to point out how the devil really is in the details, and the fact that decisions like this are being made at every level of building behavior with re: to issues like this. Simple things like this can have radical effects on the reusability of a given behavior.

Initial->Min->Max->Initial: Predetermined Percentages

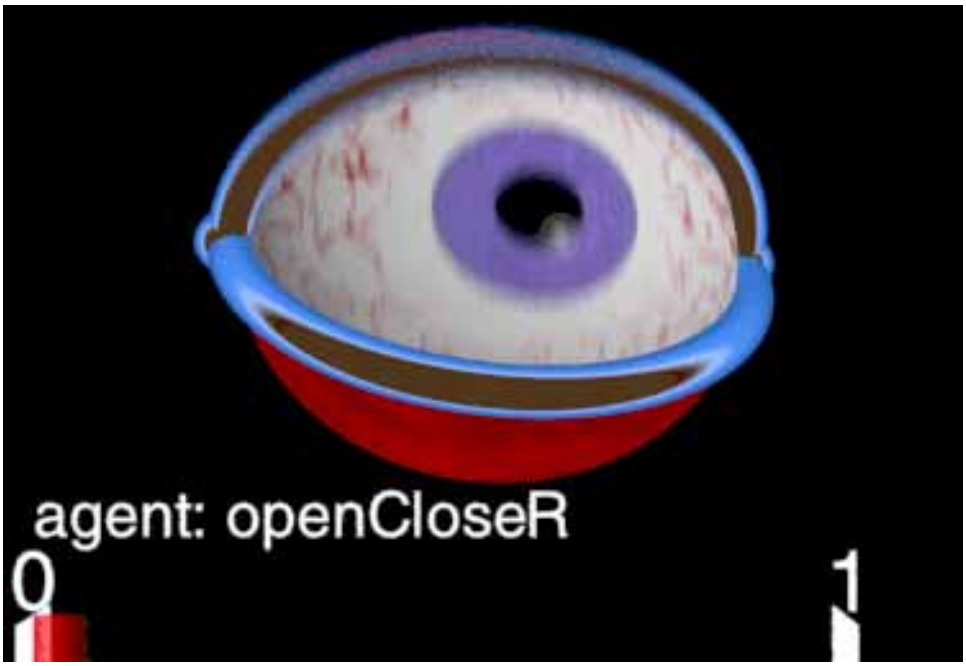
For example, let's say we implement the behavior so that for the first 1/4 of the time it goes from initial to min, for the middle 1/2 of the time it goes from min to max, and goes from maximum to the initial for the last 1/4 of the time. Then let's say that it turns out that you usually use this behavior when the pupil is in the center of the eyeball. But what happens if the pupil is initially rotated all the way over to the left? Executing the behavior in that case would result in the eye staying at the minimum for the first 1/4 of the time, then going to the max, and then whipping back to the initial point in half the time it took to go there. This is radically different behavior than we got when the initial condition was half-way between the min and max. This implementation of the behavior is terribly sensitive to the initial condition.

```
defineMotorSkill: currentMinMaxCurrent1 {varName min max duration} {

  fromCharacter: $varName


set initial [set $varName]
  set uA .25
  set uB .75
  set u 0

  while {$u <= $uA} {
    if {$uA} { set thisU [expr {$u/$uA}] } { set thisU 0}
    set $varName [expr {lerpDown($thisU, $min, $initial)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }
  while {$u <= $uB} {
    if {$uA != $uB} { set thisU [expr {($u-$uA)/($uB-$uA)}] } { set thisU 0
          }
    set $varName [expr {lerpUp($thisU, $min, $max)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }
  while {$u < 1} {
    if {$uB != 1} { set thisU [expr {($u-$uB)/(1-$uB)}] } { set thisU 0 }
    set $varName [expr {lerpDown($thisU, $initial, $max)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }

  set $varName $initial
  synchWithScene
}
```

Initial->Min->Max->Initial: Constant Velocity

On the other hand, if we looked at the total distance we're being asked to travel and moved at a constant rate, we'd have a behavior which was essentially insensitive to the initial condition.

```
defineMotorSkill: currentMinMaxCurrent2 {varName min max duration} {
  fromCharacter: $varName


  set initial [set $varName]
  set distance [expr {($initial - $min) + ($max - $min) + ($max -
          $initial)}]
  set uA [expr {($initial - $min) / $distance}]
  set uB [expr {((($initial - $min) + (1.0 - $min)) / $distance)}]
  set u 0

  while {$u <= $uA} {
    if {$uA} { set thisU [expr {$u/$uA}] } { set thisU 0}
    set $varName [expr {lerpDown($thisU, $min, $initial)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }
  while {$u <= $uB} {
    if {$uA != $uB} { set thisU [expr {($u-$uA)/($uB-$uA)}] } { set thisU 0
          }
    set $varName [expr {lerpUp($thisU, $min, $max)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }
  while {$u < 1} {
    if {$uB != 1} { set thisU [expr {($u-$uB)/(1-$uB)}] } { set thisU 0 }
    set $varName [expr {lerpDown($thisU, $initial, $max)}]
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
  }

  set $varName $initial
  synchWithScene
}
```



In this motion sequence, all 3 behaviors use the same underlying routine, which moves a given variable from an initial value, down to a min, up to a max, and back down to the initial value. While the behavior is going from initial to min, the bar is red, when it goes from min to max the bar is green, and when the behavior is going from max back down to the initial value, it is blue.

When I initially implemented the above behavior, I did it in a very specific way, with a small skill agent that knew the name of the variable in the model that we wanted to manipulate and the minimum and maximum values in degrees to modulate the value between. This worked fine for this particular model, as (say) -45 to 45 degrees were always reasonable values. But what if we wanted to reuse this behavior to:

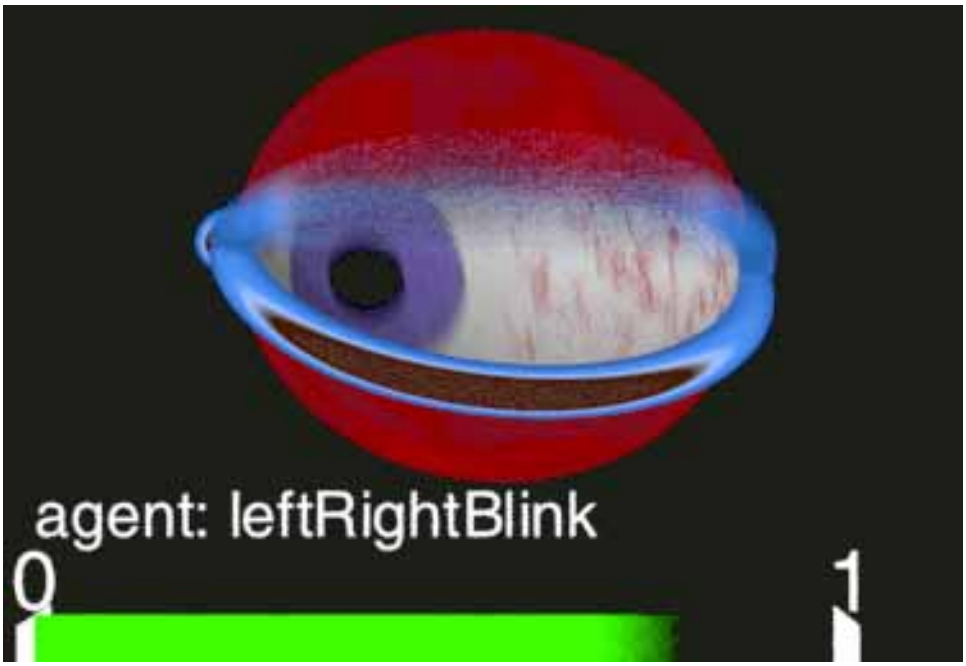
- manipulate another model's parameter where the value mapped something other than a rotation?
- i.e. moving Robbit's "eye" back in forth by translating in X.
- build a behavior that didn't go all the way down to the minimum the model supported?
- i.e. making a variation of the AudreyeII that had a weird tick that wouldn't let it rotate fully to its left.
- manipulate a model that had two variables moving in tandem?
- i.e. moving Rocky's eyes, of which there are two.
- manipulate a model that had several different variables we wanted to manipulate in this same kind of cycle (down, up, down), but at different rates?
- i.e. getting Rocky to chew his matchstick.

With the current implementation (see above) of this behavior, the first two "just work" already, since the values are normalized by the routine, and don't care what they map to, as long as a constant velocity makes sense, which it does, in the case of both rotations and translations.

The other two are potentially more difficult. Upon reflection, though, it should be obvious that the third point is actually easy; just make the inputs to the generic behavior be lists instead of scalars. One constraint that makes it much easier to code is if we constrain the velocity of the activity to be modulated by just one of the variables, say the first. Otherwise we need to write it as a set of n coroutines, which, given our current coding style, would get a lot messier.

```
proc doIt {u varName uA uB initial min max} {

   fromCharacter: $varName

  if {$u < = $uA} {
     if {$uA}
        { set thisU [expr {$u/$uA}] }
        { set thisU 0}
     set $varName [expr {lerpDown($thisU, $min, $initial)}]
  }  elseif {$u < = $uB} {
           if {$uA != $uB} \
              { set thisU [expr {($u-$uA)/($uB-$uA)}] } \
              { set thisU 0 }
           set $varName [expr {lerpUp($thisU, $min, $max)}]
     } \
     {     if {$uB != 1} \
              { set thisU [expr {($u-$uB)/(1-$uB)}] } \
              { set thisU 0 }
           set $varName [expr {lerpDown($thisU, $initial, $max)}]
     }
}


defineMotorSkill: currentMinMaxCurrent {varNameList minList maxList
           duration} {

  # sanity check the arguments; are all the lists the same length?
  if {[llength $varNameList] == [llength $minList] \
                             == [llength $maxList]}
      {error "list length mismatch" "[llength $varNameList], [llength
           $minList], [llength $maxList]"} {}
  set howMany [llength $varNameList]

  # we need to do this to get the initial value(s) of
  # the variable(s) in varList
  for {set i 0} {$i <  $howMany} {incr i} {
     fromCharacter: $varName
  }

  for {set i 0} {$i <  $howMany} {incr i} { \
    set initial  [set [lindex $varNameList $i]]
    set min      [lindex $minList $i]
    set max      [lindex $maxList $i]
    set distance [expr {($initial - $min) + ($max - $min) + ($max -
           $initial)}]
    set uA [expr {($initial - $min) / $distance}]
    set uB [expr {((($initial - $min) + (1.0 - $min)) / $distance)}]
    lappend initialList $initial
    lappend minList $min
    lappend maxList $max
    lappend uAList $uA
    lappend uBList $uB
  }

  set u 0

  while {$u <  1} {
    for {set i 0} {$i <  $howMany} {incr i} {
       set varName [lindex $varNameList $i]
       set initial [lindex $initialList $i]
       set uA [lindex $uAList $i]
       set uB [lindex $uBList $i]
       set min [lindex $minList $i]
       set max [lindex $maxList $i]
       doIt $u $varName $uA $uB $initial $min $max
    }
    set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
           [synchWithScene])}]
  }

  for {set i 0} {$i <  $howMany} {incr i} {
     set varName [lindex $varNameList $i]
     set initial [lindex $initialList $i]
     set uA [lindex $uAList $i]
     set uB [lindex $uBList $i]
     set min [lindex $minList $i]
     set max [lindex $maxList $i]
     doIt 1 $varName $uA $uB $initial $min $max
  }
  synchWithScene
}
```

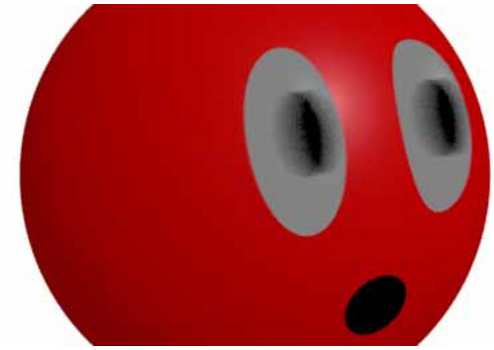
For example, we could use this motor skill to generate a behavior where AudreyeII looked left and right as she blinked:



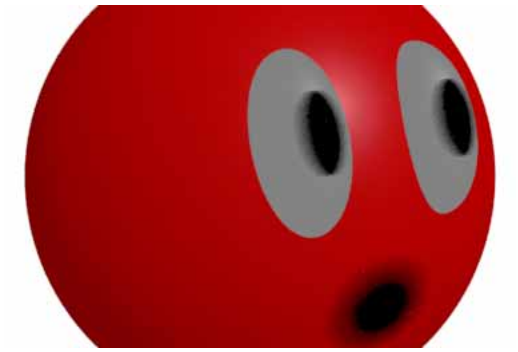
So now that we've built a reusable, reasonably general skill agent that can generate useful behavior, let's think about where we could reuse it.

Reuse on SphereHead

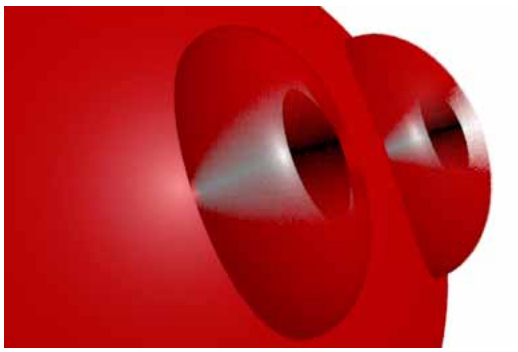
SphereHead has several articulated variables that could use this. For example, we could attach it to the two variables that control where his eyes are looking, and allow him to look left and right:



Or we could also attach it to his mouth, allowing him to gasp in surprise when he noticed some particular occurence:

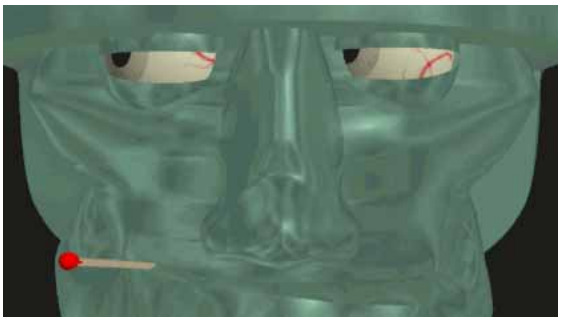


Of course, we could also let him blink using this behavior:

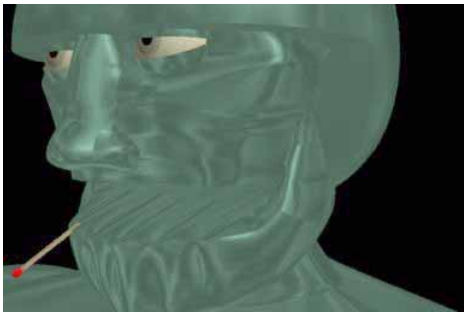


Reuse on Rocky

Rocky could also use this to look left and right, either just by shifting his eyes, or by moving head head. Also, Rocky can blink too.



Also, Rocky's mouth can move about, so you can use this to make a chewing behavior (although he has no teeth...):



Reuse on Robbit

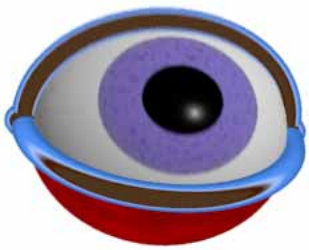
Robbit also has a large number of articulated variables that can be manipulated nicely with this behavior. We can use it to have his glowing eye move back and forth, have his head move around, or move his body around, move up and down, or wag his tail back and forth or up and down.

Building Behaviors by Composition

In the last chapter, we saw how we could compose models together out of other models. In the same way, we can compose agents together to build up more complex behavior. For example, let's say we wanted to take the simple behaviors we wrote in the previous section and use them to build a more complex behavior for Audreye, perhaps one where she is reacting to a bright light. We might look at a pupil constrict behavior, a blink, and a rotate, and compose them into a "react to light" behavior.

Here's a behavior "composed" of several simpler ones. This is a behavior in response to a light being shined at Audreye. We want the pupil to dilate, blink twice, have the pupil relax a bit near the end, and Audreye should rotate away from the source. Finally, we make the eye get a bit more irritated; i.e. bloodshot. The main loop of the skill agent's motor skill looks like this:

```
set u 0

 # let's identify what to do when
 # we'll be blinking twice over the entire span

 set thetaI $audreyeII(thetaMax)
 set thetaDif [expr {360 - $thetaI}]

 set irisI $eyeBall(pupilSize)
 set irisM .01
 set irisF .03

 set bloodI $eyeBall(bloodShot)
 set bloodF [expr {$eyeBall(bloodShot) * 1.5}]

 set eyeI $yR; # this is normalized from 0 to 1
 set eyeF [expr {$yR - 30}] ;# we want to go down...

 while {$u <.75} { set eyeBall(bloodShot) [expr {lerpUp((($u - .75) * 4),
          $bloodI, $bloodF)}] } {}

   # bring the pupil back open a bit at the end
   if {$u > .8} { set eyeBall(pupilSize) [expr {lerpUp(($u * 5), $irisM,
          $irisF)}] } {}

   set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
          [synchWithScene])}]
 }
```

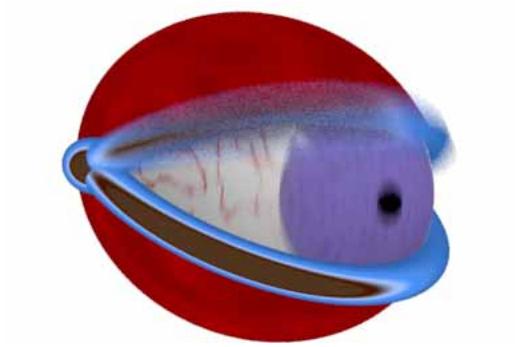


There are problems with that behavior, though. The blink takes too long; so let's replace the line:

```
    set audreyeII(thetaMax) [expr {$thetaI + ($thetaDif * abs(sin($u * 2 *
           pi())))}]
```

with:

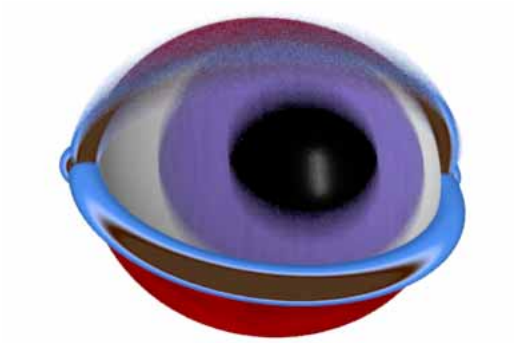
```
if {$u < = .1} { set audreyeII(thetaMax) [expr {$thetaI + ($thetaDif *
        abs(sin( ($u * 10) * pi()))}]} {}
if {$u > = .9} { set audreyeII(thetaMax) [expr {$thetaI + ($thetaDif *
        abs(sin( (($u - .9) * 10) * pi()))}]} {}
```



One thing to notice here is that this part is much more prone to aliasing, as it is happening in a much smaller span of time.

Let's try yet another version, where the main body of the skill agent looks like this:

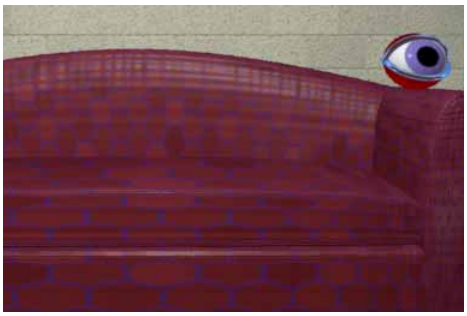
```
set u 0
# let's identify what to do when
# we'll be blinking twice over the entire span

set thetaI $audreyeII(thetaMax)
set thetaDif [expr {360 - $thetaI}]

set irisI $eyeBall(pupilSize)
set irisM .03
set irisF .05

set bloodI $eyeBall(bloodShot)
set bloodF [expr {$eyeBall(bloodShot) * 1.5}]

set eyeI $yR; # this is normalized from 0 to 1
set eyeF [expr {$yR - 50}] ;# we want to go down...

while {$u < 1} {

  # deal with the two blinks
  if {$u <= .1} { set audreyeII(thetaMax) [expr {$thetaI + ($thetaDif *
        abs(sin( ($u * 10) * pi()))}]} {}
  if {$u >= .9} { set audreyeII(thetaMax) [expr {$thetaI + ($thetaDif *
        abs(sin( (($u - .9) * 10) * pi()))}]} {}

  # now do the pupil constriction quickly
  if {$u < 1./3.} { set eyeBall(pupilSize) [expr {lerpDown(($u * 3),
        $irisM, $irisI)}]} {}

  # rotate the eye and the eyeBall
  if {$u < .5} { set yR [expr {lerpDown(($u * 2), $eyeF, $eyeI)}]} {}

  # make him a bit more bloodshot at the end
  if {$u > .75} { set eyeBall(bloodShot) [expr {lerpUp((($u - .75) * 4),
        $bloodI, $bloodF)}]} {}

  # bring the pupil back open a bit at the end
  if {$u > .8} { set eyeBall(pupilSize) [expr {lerpUp(($u * 5), $irisM,
        $irisF)}]} {}

  set u [expr {$u + (1.0/(($duration * $scene(ticksPerSecond)) - 1) *
        [synchWithScene])}]
}
```

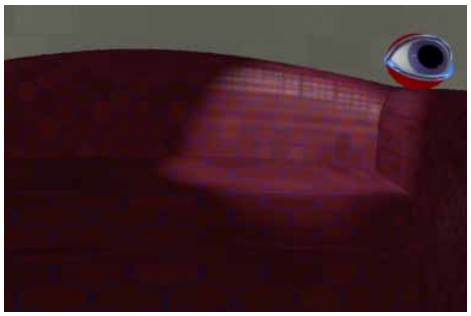


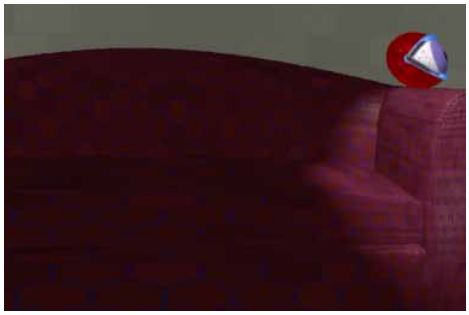
Audreye in the Room

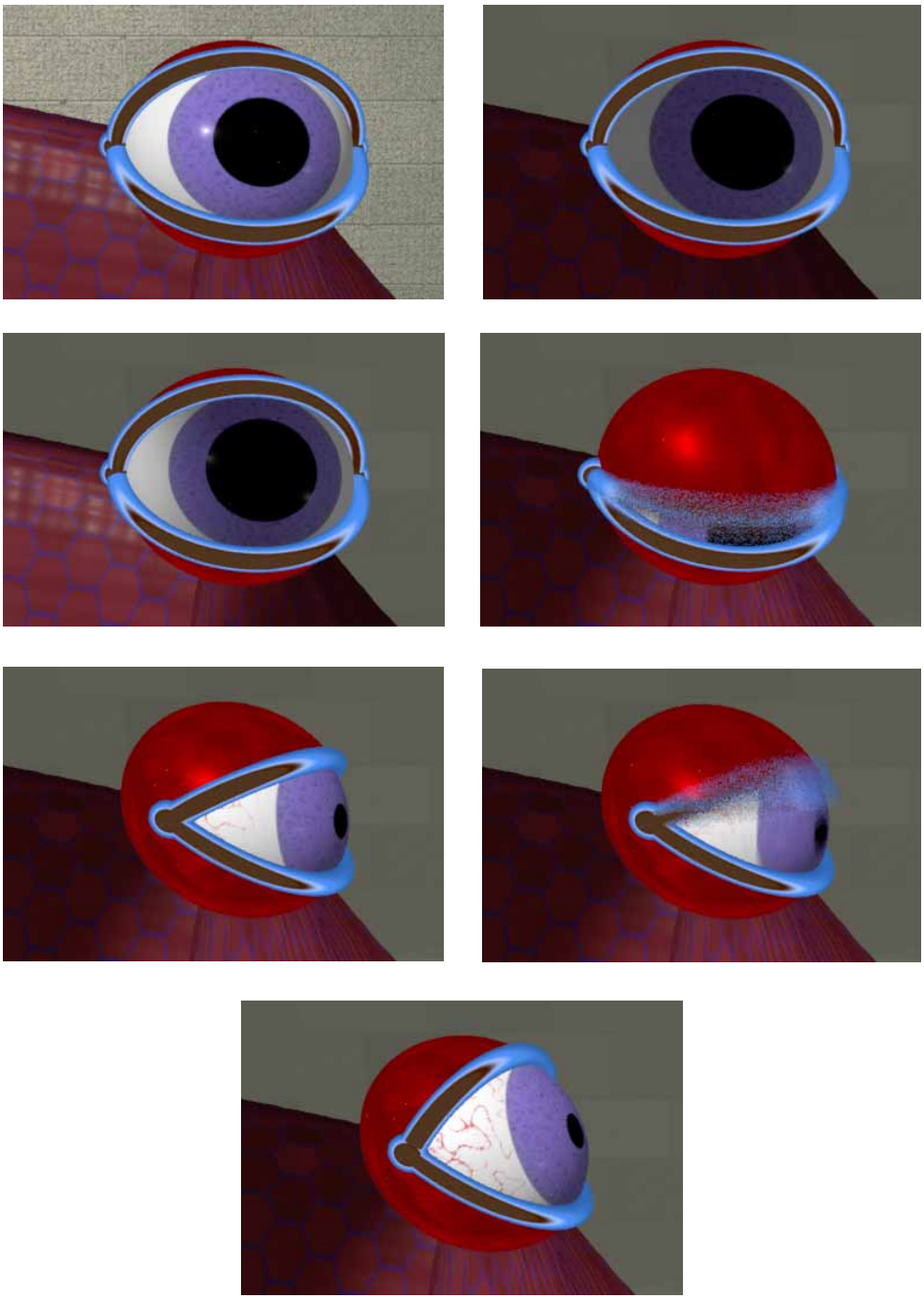
Recall the room I described in the previous chapter:

One interesting property of this room is the fact that the spotlight in it (the only direct light source in the room), is, for all intents and purposes, completely described by 3 articulated variables: the angle of the track light it's part of, the intensity of the light, and the width of the spotlight's cone. By building a sensor agent that has receptors for those three variables, we can build a sensor agent **`inSpotLight`** to let a character know if it's in the spotlight. We could then attach this sensor agent to the skill agent we described above. Here's what we would see, from two different vantage points:









This is a perfect example of a useful, interesting behavior that is too simple to try to configure in with a larger action selection network, but is perfectly amenable to a more "messy" approach, where we just directly wire the sensor agent and the skill agent into the character. We can tune the reactivity of the behavior (is it very sensitive, is it sluggish, etc.) by changing the sampling rate of the various receptors attached to the sensor agent. We can tune the execution of the behavior by manipulating the parameters of the skill

agent; how long does it take to turn, how bloodshot does the eye get, how much does the pupil constrict, etc.

Once we're happy with that behavior, we could begin layering on others; perhaps a sensor agent that notices slight differentials in the amount of light, or perhaps a long running skill agent built as a servo-mechanism which is always trying to keep the diameter of the pupil in synch with the amount of light falling on the eye. The possibilities, even with this rather simple character, are endless...

Conclusion

In this chapter, we saw the power and flexibility that we get by thinking about animating characters by building and composing agents together and having them act on the articulated variables of a model over time. Using this agent-based approach, we can integrate both sophisticated reactive planning algorithms like Maes' spreading activation planner with simpler behavior-generating agent configurations like a single sensor agent and skill agent. This approach gives us a way of building reusable, composable, blendable behaviors. When coupled with the fact that the eve object framework allows us to record everything that happens, we've moved several steps closer to the dream of being able to build characters.