Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science

Proposal for Thesis Research in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Title: A K-Line-Based Language for Artificial Intelligence Programming

Submitted by:	Robert A. Hearn 11 Story St. #32 Cambridge, MA 02138	____________________ (Signature of author)

Date of submission: December 16, 2002

Expected Date of Completion: December 2004

Thesis supervisor:	Gerald Jay Sussman	Matsushita Professor of Electrical Engineering, MIT
Thesis reader:	Marvin Lee Minsky	Toshiba Professor of Media Arts and Sciences Professor of E.E. and C.S., M.I.T
Thesis reader:	Patrick Henry Winston	Ford Professor of Artificial Intelligence and Computer Science, MIT

Brief Statement of the Problem:

Minsky's book "The Society of Mind" offers a set of powerful insights on how one might go about constructing an artificial mind. To put these ideas into practice, there are several engineering challenges that must be overcome. Chief among these is designing an explicit memory architecture that adequately supports the concept of K-line. I propose to design and implement a programming language based on K-lines. The language will be declarative, specifying something akin to a connectionist network, which is then simulated.

I will demonstrate the power of this language by using it to program simulated creatures, inhabiting a two-dimensional world. The runtime environment for the language will thus consist of a robust two-dimensional physics simulator, as well as the network simulator. The creatures will be able to perform such tasks as walking, picking up and manipulating objects, navigating with vision and memory, solving problems, communicating with each other, and learning new behaviors.

1 Introduction

Artificial intelligence (AI) has been an active area of research for nearly fifty years, yet the original goal – computational human-level intelligence – seems as far off as ever. To a large degree the size of the problem was initially underestimated, but over the past couple of decades the focus of AI research has shifted away from attempting to understand and build truly intelligent systems. Yet, I maintain that the keys are there, if one knows how to find and use them. In 1986, Marvin Minsky published "The Society of Mind" [15]. This book lays out a grand vision of human cognitive architecture. It is the inspiration for the programming language I will design for building intelligent systems.

In the Society of Mind (SOM) model, a mind is seen as a large society of interacting "agents", which use each other to accomplish their tasks. Some agents directly control motor function, some report sensory input, but most perform higher-level functions. Agents are organized into "agencies", groups of agents which collectively perform complex behaviors and represent knowledge of various kinds. Many different structuring and connection paradigms are proposed to enable different sorts of processes. Examples include *K-lines*, *frames*, *polynemes*, *micronemes*, *pronomes*, and *isonomes*. Of these, K-lines are the most fundamental, as they are the basic units of memories. Thus, they are the concept around which the programming language I propose is centered.

A New Language

Why is a new programming language necessary? AI researchers have been using Lisp from time immemorial, and with the wide variety of languages available today, surely almost any need may be met by an existing language. But Lisp and other languages lack the primitive notions needed for programming in a SOM style. By and large, programming languages most naturally express serial processes, but to program in a SOM style, one needs thousands of different, mutually interacting subprograms. Of course any program may be written in any language, but a suitable language significantly simplifies and better directs the programming task. The key issue is what is implicit vs. explicit in a language. In the K-line-based language I propose, massive concurrency and appropriate memory models are implicit, but many ordinary computer science concepts (e.g. recursion, arithmetic operations) are not.

As a runtime system supporting this programming style, I propose a "cognitive simulator" which is a kind of augmented connectionist network. Connectionism attempts to model mental processes in terms of interactions of large numbers of simple processing units, e.g. neural networks [3]. Typically each unit has a numerical "activation level", which affects the activation levels of the units it is connected to as time evolves. Often the weights connecting the units also evolve according to learning rules. The networks I simulate will have additional built-in memory capabilities vs. a conventional neural network: all of the memory-management microstructure will be abstracted into higher-level capabilities for manipulating K-lines, rather than implemented directly as large assemblages of primitive units.

A New World

Why is a simulated world necessary? Again, AI researchers have been programming AI systems in abstract domains from time immemorial. Again, true intelligence has not yet been achieved. In the 1980s the situated intelligence movement arose [2, 19], with the central tenet that human intelligence is inherently a phenomenon involving interaction with an external reality. In its most extreme form, "embodied intelligence" [18], this doctrine holds that this must be literal physical reality; a simulation is not good enough. Thus only by building robots could one hope to achieve true artificial intelligence. However, just like the "Good Old Fashioned AI" (GOFAI) the situated movement denigrates, there does not seem to be an obvious path to true artificial intelligence on the embodied front either. I reject the extreme embodied doctrine, but find some valid points to the general situated point of view. We know intelligence is possible in systems (people) that interact with an external, continuous environment, on a real-time basis. Intelligence of the pure symbolic GOFAI sort may be possible as well, but it must be admitted that most such systems designed to date are not structured very much like the biological systems they are inspired by. It surely cannot hurt, and can possibly help, to give our programs a runtime context that is closer to the one experienced by people. But building actual robots is very time- and labor-intensive, and requiring actual physical domains restricts the possible contexts for the systems to be immersed in. A simulated world enables systems that interact with their environment in a realistic manner, but that may be run and tested much more conveniently than real robots.

The next question that must be asked is, why simulate a two-dimensional world? After all, people live in three dimensions. Simulating a two-dimensional world is simpler in several respects, however, and a two-dimensional world can still provide the kind of analog external environment we are seeking to help structure our programs. First, a two-dimensional physical simulation is more computationally efficient than a three-dimensional one. Second, it simplifies the design of various low-level creature behaviors, such as walking without falling over. Such tasks are interesting in their own right, but are presumably not at the heart of intelligence (though some embodied AI adherents may disagree). In a similar vein, processing vision from a one-dimensional retina is much more efficient than processing vision from a two-dimensional retina. The computational savings will lead to the ability to simulate more creatures of greater complexity, in a more complex environment, than would be possible with a three-dimensional simulation. It might be asked whether a two-dimensional world is sufficiently rich in behavioral possibilities for the simulated creatures. A. K. Dewdney's book "The Planiverse" [9] provides an answer in the affirmative. Dewdney sketches a two-dimensional world, peopled with beings with an appropriate anatomy for my creatures, and filled with all manner of devices, structures, and buildings. Though fictional, the book adequately illustrates that interesting problems may be solved and adventures had in such a world. I use the Planiverse as a model for the world I will simulate.

Related Work

This work follows directly from work done for my Master's thesis, "Building Grounded Abstractions for Artificial Intelligence Programming" [12]. That project was, to my knowledge, the first attempt to take many of the architectural ideas from SOM at face value and construct an architecture to simulate them. There are three ways that this proposal extends that work. First, there was no explicit programming language; the creature networks were hand-coded in C++. Second, the simulated world was very simple, and not physically robust. Third, the implemented creature behaviors were fairly rudimentary.

There has been at least one other programming language inspired by ideas in SOM. Framer [11] (and its descendent, FramerD) is a language based on the concept of *frame*, which is a key knowledge representation structure in SOM. Framer is mainly designed to manage large knowledge bases and semantic networks. It does not directly serve to specify behavioral programs, as the language I propose does.

Michael Travers' LiveWorld [21] is a programming environment (based on Common Lisp) designed to support the construction of animate systems. Travers' "agent-based programming" is inspired by SOM, although the emphasis is different. In [21] the agent metaphor is used for interaction of mobile agents in a shared physical environment – e.g., an ant colony. In this proposal, by contrast, a large network of agents controls a single creature; the emphasis is on the architectures for behaviors of an individual mobile entity.

Overview

There are four main components of what I plan to accomplish: (1) the K-line-based language itself, (2) the cognitive simulator that runs the programs, (3) the physics simulator that provides a runtime environment, and (4) the creatures programmed with the language. Ultimately, the level of behavioral sophistication attained by the creatures will be the measure of this work.

These components are described in more detail in the following sections:

- Section 2 discusses the memory models that are needed for the language. The key issue is how to represent working memory.
- Section 3 presents some ideas for details of the programming language. I give a preliminary syntax with examples of how some programs would get translated into networks to simulate.
- Section 4 describes the two-dimensional world I plan to simulate.
- Section 5 lays out my plan for the layers of increasingly complex behaviors the simulated creatures will have.
- Section 6 describe the cognitive and physics simulators.

2 Memory Models

The design of my language must enable the kinds of agent interactions that characterize SOM. The principal paradigm of interaction, unlike in other multi-agent systems, does not involve an elaborate communication protocol, or a structured "agent communication language". Although agents must use other agents to achieve their desired goals, and must learn how to do so, this learning does not lead to a language per se. Instead, the standard paradigm for agent A using agent B to accomplish a task is first to set some other agent(s) C into a particular state, then simply to activate B. The state of the C agents specifies the details of the desired behavior. This may seem like an arbitrary semantic distinction from sending a more traditional kind of message, but it is not. The key is that the C states have meaning in and of themselves; they are an existing part of the system, on a par with A and B.

As an example, suppose there are some visual state-representing agents (collectively, an *agency*), that get put into a particular state when an apple is perceived. An agent whose task is to pick up an apple might first put this agency into an *apple* state, causing it to "hallucinate" an apple, then activate a grasping agent. The grasping agent would not need to be told what to grasp; the visual agency will take of that automatically. It will be predisposed to notice any apples in the present scene, because it has "just been thinking about apples". The visual state agency does not exist as a mere communication buffer, but has an intrinsic function of its own. Importantly, the representation chosen for the "message" is dictated by the existing structure of the visual state agency.

The mechanism by which an agent puts other agents into particular states is called a *K-line*. (*K* is for *knowledge*.) A K-line is like a snapshot of a previous total or partial activation state. In the SOM model (and, originally, in Minsky's earlier K-lines paper [14]) K-lines are formed when memorable events occur, on the premise that a record of the then current mental state is likely to be useful at a future time. When a similar partial state arises later on, the situation can be recognized as related to a previous one, and the appropriate K-line activated. In the example above we imagine that the visual agents have learned that the states that result when an apple is seen are worth remembering, and have formed a K-line. Likewise, the picking-up agent has learned that activating the apple K-line is useful for achieving its ends. K-lines form the foundation of the memory system needed to support a SOM architecture. Figure 1 shows the conceptual structure of K-line activation and recognition. (Recognition can also be tied to activation, for a "ring-closing effect" [15, Section 19.10], either directly, or at a higher level.)

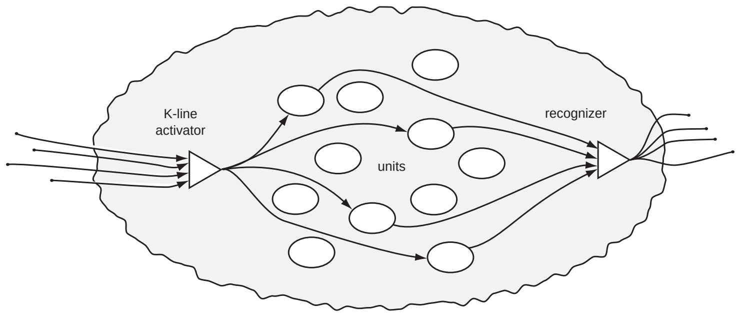


Figure 1: An agency composed of units.

In the above apple-grasping example, the correct behavior depends on the details of how the visual state-representing agents' behavior is affected by their recent states. In other words, it depends on the details of the "short-term" or "working" memory structures. Similar issues arise when more than one pair of agents might be trying to communicate using the same intermediary agency. In SOM parlance, working memories are called *temporary K-lines* [15, Section 22.1]. There is no detailed mechanism proposed in SOM for how temporary K-lines might work, but a model of their capabilities is needed in order to design the language. I will explore a couple of different possibilities, explained below.

2.1 Model 1: Temporary K-lines are Like Permanent K-lines

The simplest idea for how temporary K-lines might work is that they are just like permanent K-lines, except that they can be formed very rapidly. One can imagine that all of the recent interesting states of an agency are recorded as K-lines for later re-use, either short-term or long-term. I won't speculate on the neural structures needed to support such a system in a real brain.

How can this sort of temporary K-line work to enable the apple-finding behavior described above? One useful trick is to tag temporary K-lines with identifiers specifying the role they are intended to serve. Suppose that when the apple-picking-up agent activates the *apple* K-line, it also activates another, special agent, which I'll call *target*. The visual state agency forms a new temporary K-line, which records both the *apple* activation and *target*'s activation. After the K-line has formed, signals flooding into the visual state agency from the eyes induce other states, based on what is currently seen. But when an apple is seen, the temporary K-line is recognized and activated, complete with the *target* agent. Now, all the grasping agent has to do is wait until *target* turns on again; then it knows that the desired object (whatever that may be) is now the focus of visual attention, ready to be grasped.

Target is an example of what is referred to in SOM as a *pronome*. It is an agent that serves to indicate the role a particular memory should play. We can imagine more complex behaviors, that need multiple pronomes. For example, to put the apple into a bag, it would be nice to use a generic "put something into something else" agent, and control it with temporary K-lines. But both somethings could need to be represented using the same agency. We can tag the apple memory with *target*, and the bag memory with *destination*. Then to control this agent, we simply form temporary K-lines for *apple* + *target* and for *bag* + *destination*, then activate the putting-into agent. Many other agents and agencies may need to be involved in achieving the overall task, but we can use more pronomes and temporary K-lines to communicate between them.

These K-lines are temporary in the sense that they can be re-used, effectively erasing earlier memories. Maybe the next time the grasping agent is invoked, *target* is a glass of water rather than an apple. For the behaviors to function correctly, the *target* pronome must reactivate when a glass of water is seen, but not when an apple is seen. There are various ways of achieving this effect, but typically they involve the temporary K-lines weakening over time. Conceivably *all* K-lines weaken over time, unless they are strengthened and made permanent by frequent reactivation, or some other process. Since the high-level behavior depends on the low-level details of the weakening process, it could be problematic to engineer robust programming abstractions based directly on this model of temporary K-lines.

A related problem is *frame recognition.* A *frame* in SOM is a structure that represents some particular kind of situation, independent of specific details of that situation [13]. Examples include an office, a dinner party, and tying one's shoes. In each case, we need to recognize when the given situation obtains, in order to apply the needed behaviors for that situation. But this is not as simple as recognizing a simple object, like an apple. For example, to recognize that we are in an office, we might notice that there's a desk on the floor, a white board on the wall, and a computer on the desk. If the same agencies are used to represent each of these objects, then in a sense we can only be "currently aware" of one of them at any instant. Then how can we detect their simultaneous presence, to activate the office frame? We can start by using the pronome trick again. Maybe when we see a white board on the wall, we form a temporary K-line for *white board* + *on the wall*, and similarly for the desk and the computer. *On the wall*, *on the floor*, etc. then act as pronomes, indicating the roles played by the objects associated with them. Then all the information needed to recognize the *office* frame is present in temporary K-lines. But in order to actually recognize the *office* frame, what is needed is for the K-line recognizers for *white board* + *on the wall* etc. to remain active for a while, even when the K-line contents are no longer being perceived. Otherwise, all the needed inputs for the *office* recognizer will not be active simultaneously. Again, we see that correct behavior depends on low-level details of the activation level behavior of some units.

One further problem, which perhaps the reader has already seen, is that an agency's current function might get disrupted when it is being used for building temporary K-lines with pronomes. If a vision agency is looking at a glass of water when another agent activates its *apple* K-line, won't that confuse behaviors that depend on the current contents of the vision agency? We can suppose a further mechanism that keeps the outside world from noticing while temporary K-lines are being assigned, but then we might have to worry about the details of that mechanism.

I suspect that something like the above model is close to how most of our working memories actually function. But the difficulties in dealing with such a model directly suggest we should try other models as well. I plan to explore this model further, trying to come up with abstractions that can be expressed as programs, but currently my language is expressed in terms of the following model.

2.2 Model 2: Temporary K-lines are Explicit Mirrors of Agency State

One way to solve the problems with the simple temporary K-line model is to assume that temporary K-lines are literal copies of the source agencies they are attached to. Thus, they are fundamentally different things from permanent K-lines. For each pronome needed, we assume a copy – a parallel buffer of "mirror" units – of each source agency which uses that pronome. This is illustrated in Figure 2. Then to store, e.g., *apple* + *target*, we simply activate *apple*, and store the current state into the attached *target* copy. This is the approach to working memories developed in [12].

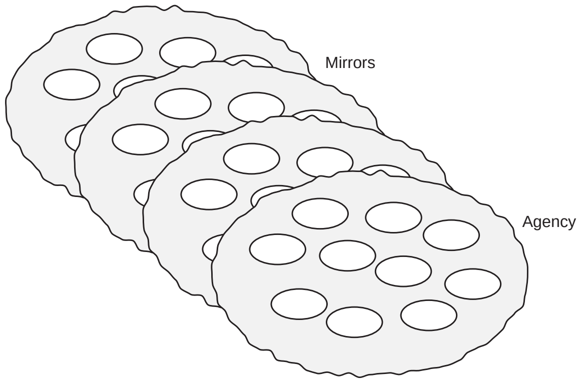
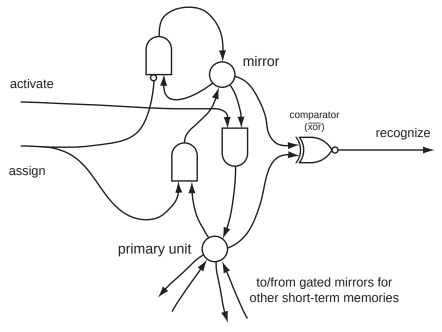
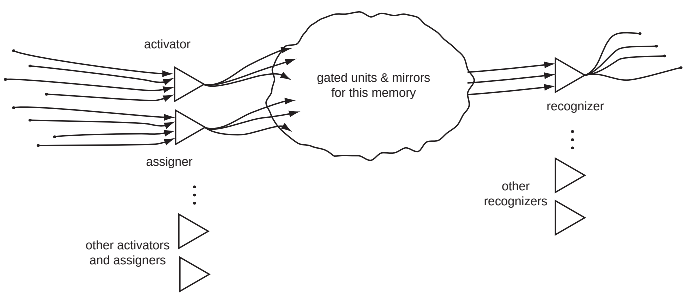


Figure 2: Working memories as mirrors of agencies.

To store and reactivate the temporary K-lines, we need some simple control logic for each unit / mirror pair, shown in Figure 3(a). *Assign* gates the primary unit's activation into the mirror unit, and *activate* gates it back. *Recognize* detects when the two units have the same value. (This particular circuit assumes binary activation levels, but a slightly more complicated circuit will work for arbitrary activation levels.) The structure for an entire working memory copy is shown in Figure 3(b). The *activate* etc. units branch to one copy of the control logic for each unit / mirror pair. Note that we need to be able to recognize when the working memory state matches the current agency state in order to perform some of the tasks described above.


mirror
activate
assign
comparator
($\overline{\text{xor}}$)
recognize
primary unit
to/from gated mirrors for
other short-term memories

(a) Mirror logic for a single unit



(b) A working memory for an agency

Figure 3: Control logic for mirror-style working memories.

The model as described so far solves the memory erasure problem mentioned above. Since each working memory can have only one set of current contents, there is no possibility of confusion or fuzziness about what its current value is. To solve the other two problems mentioned, we need to add some more capabilities. Since working memories are copies of agencies, it is reasonable to suppose that, like agencies, they have machinery for forming their own K-lines. This concept seems a little strange – a short-term memory having its own long-term memories? But it is really quite natural, and useful. In the simple model, we had a problem forming temporary K-lines, in that their formation might interfere with the normal agency function. In this model, there is no problem; to put *apple* into the *target* memory, we simply activate a K-line of the *target* memory itself, bypassing the visual agency entirely.

Similarly, to recognize frames, all we need is K-line recognizers in the working memories. For the office example above, if we have seen a white board on the wall recently, we can represent this by putting *white board* in the *on wall* memory. Then, the office frame recognizer can simply recognize the simultaneous activation of a sufficient set of relevant working memory recognizers. This resolution of the so-called *frame recognition problem* [15, Section 24.9] is one of the chief contributions of [12].

At this point the careful reader will notice an apparent problem. We are assuming that, for example, a visual attribute-representing agency has a K-line for *apple*, that puts it into a particular state (specifically, activates some subset of its component units). Furthermore, a working memory copy of the agency, *target*, also has its own *apple* K-line, which happens to activate exactly the corresponding units. How could such a structure arise? This seems to call for duplication of learned structures, which is not biologically plausible. However, this problem is illusory. Just as the visual attribute agency has learned that *apple* is a useful state, and created a K-line for it, the *target* memory has also, independently, formed its own K-line, presumably because apples are often desired as target objects. *Target*'s version of the K-line activates the corresponding units, because those are the ones that are active when the K-line gets stored there from the source agency. It is *target*'s K-line that the apple-desiring agent has learned to use.

A more serious problem is that each agency requires a working memory copy for each pronome it can use. Presumably, behaviors that will not interfere can use the same memories to represent different pronomes, but even so, how many is enough? Biological considerations suggest it should be a small number, say half a dozen. That is roughly the number of distinct "items" a person can keep in mind at once, although it is hard to tell exactly what this means. It is not clear that half a dozen is a sufficient number for the uses we will need. A related problem is that pronome-like associations are of an entirely different form from other memory associations, e.g. the fact that apples are red and round. Intuitively, recognizing an office by seeing a white board on the wall and a desk on the floor should be similar to recognizing an apple by noticing that it is both red and round. It is for these reasons that the model in Section 2.1 seems more plausible to me, but the mirror-based model has the necessary virtue of ease of abstraction. Perhaps there is a way to simulate the mirror-based model with the simple model; this is something I plan to investigate further.

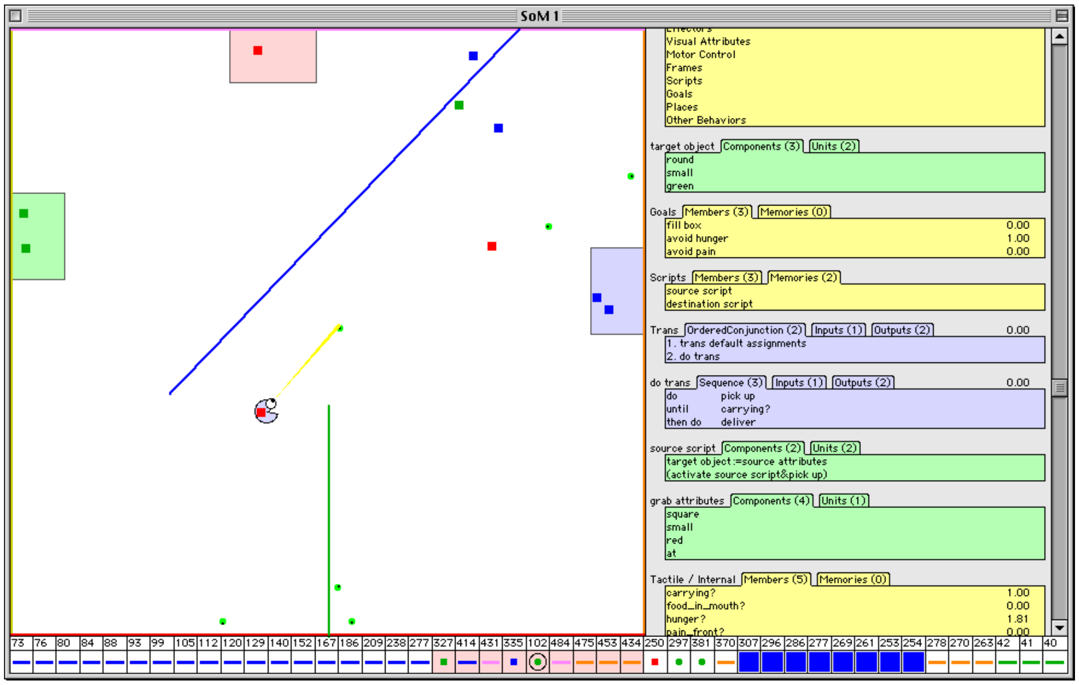
Memory Abstraction

When one specifies a program in a programming language, it is best to be able to use high-level abstractions. Programming creature behaviors at the level of individual units, as shown for these memory architectures, is like programming in assembly language. In [12] I showed that the kinds of memories described in Section 2.2 may be treated abstractly, and specified and even simulated at a higher level. The fundamental operations in the current form of the language I am proposing involve creating, activating, and reassigning these working memories, and permanent K-lines. The multitudes of units composing the mirrors do not need to exist explicitly in the cognitive simulator; instead, the simulator can dynamically "rewire" infividual units that serve the same role, in a manner that preserves the behavior described here.

3 The Language

The programming language allows specification of behaviors in terms of basic K-line relationships and operations. The result is a connectionist-style unit network, which is then simulated by the cognitive simulator. The programming language is declarative; programs written in it simply define the initial network state. The memory operations that occur as the network runs are controlled by the structure of the network itself. Over time, the network can structurally change, as new memories are formed and working memory states change. In essence the programs are self-modifying. There is no distinction between program and data.

I have not designed the language in detail yet, but I have written some sample code in a form which might resemble the ultimate language. For illustrative purposes, I have translated some creature behaviors used in my Master's thesis [12] into this proto-language. (In [12] the behaviors were hand-constructed in C++.) The creature implemented in [12] also inhabits a two-dimensional world, although a much simpler one than the more realistic one planned in this proposal. In the simple world, there is no gravity; the creatures wander around on a plane, viewed from above. There is no realistic physics simulator; all creature-world interactions are very basic. The creature can turn and move in a continuous space, atomically pick things up, put them down, eat things, and run into walls. It has three goals in life: one, pick up colored blocks and put them into matching boxes; two, eat when it gets hungry; three, avoid pain (caused by running into walls). The creature is shown in its habitat in Figure 4.



```
Visual Attributes
Motor Control
Frames
Scripts
Goals
Places
Other Behaviors

target object  Components (3)  Units (2)
round
small
green

Goals  Members (3)  Memories (0)
fill box                                   0.00
avoid hunger                               1.00
avoid pain                                 0.00

Scripts  Members (3)  Memories (2)
source script
destination script

Trans  OrderedConjunction (2)  Inputs (1)  Outputs (2)     0.00
1. trans default assignments
2. do trans

do trans  Sequence (3)  Inputs (1)  Outputs (2)            0.00
do        pick up
until     carrying?
then do   deliver

source script  Components (2)  Units (2)
target object:=source attributes
(activate source script&pick up)

grab attributes  Components (4)  Units (1)
square
small
red
at

Tactile / Internal  Members (5)  Memories (0)
carrying?                                  1.00
food_in_mouth?                             0.00
hunger?                                    1.81
pain_front?                                0.00
```

Figure 4: A simple creature in its world.

Example Behaviors

Before going into the language details, I will first present and explain some sample behavior networks, shown in Figures 5 and 6. The nodes in these network denote the basic computational units (including those serving as K-lines), and the arcs denote the connections. (Arrows on the arcs are omitted; the direction is usually clear from context.) Sometimes the arcs are labeled; in this case they refer to the specific working memory to which the activation should be directed. The graph notation is described more fully in [12].

I use the memory model described in Section 2.2, where working memories are implemented as "mirrors" of their source agencies. I also plan to explore the other working memory model discussed (Section 2.1), where working memories are not distinguished from ordinary K-lines. As mentioned there, the mirror model seems to lead to simpler programmatic expression; I haven't yet fully explored how to specify behaviors using the other model.

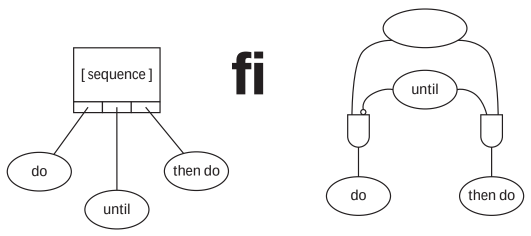


Figure 5: Sequence construction.

Figure 5 shows how to sequence behaviors. The form on the left is shorthand for the structure on the right, which activates the unit labeled *do* until *until* becomes active, and then activates *then do*. This is similar to an `if-then-else`, but the usage is different; both branches are expected to be performed. When the sequence unit is active, the sequencing behavior is "running", controlling the activity of the behaviors to be sequenced.

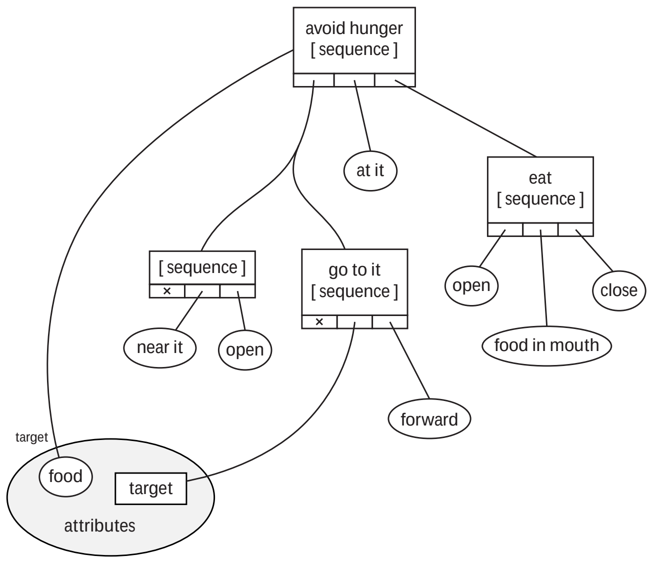


Figure 6: Avoid hunger construction.

Figure 6 shows the structure of the creature's behavior for avoiding hunger. (This behavior's activation would be controlled by a higher-level agency managing goal priorities.) Here the sequence structure is used as a kind of macro. This network first loads the *food* memory into the *target* working memory of the *attributes* agency. *Attributes* is the agency that represents visual state, and *food* is a K-line that activates the state the agency would be in if the creature were perceiving food. Here *food* is activated directly into *target* rather than into the agency itself.

The *avoid hunger* sequence runs a *go to it* behavior until *at it*, then runs an *eat* behavior. While going to it, it also runs a behavior that opens its mouth if it is near food. The *target* memory is considered to be active when its state matches that in the *attributes* agency, i.e., when the target object is being perceived. Here, this triggers forward motion by the *go to it* behavior. *Near it* and *at it* are units that are recognize when *target* is active, in addition to *attributes* indicating the appropriate proximity. Other simple behaviors (not shown) cause the creature to automatically direct its eye and its body toward things matching *target*.

This simple network serves to coordinate an appropriate series of behaviors to keep the creature from starving. The *avoid hunger* behavior itself is activated when the hunger level reaches a threshold value (which is opportunistically lowered if food happens to be seen nearby). The key to this network"s operation is the ability of the *target* working memory to be directly addressable just as its source agency is, and to automatically detect when its state matches.

Example Programs

Now, let's see how these behaviors might be represented in my language. We begin with the top-level specification of *avoid hunger*:

```
avoid_hunger =
[
    attributes: target: food;

    sequence(
        [ go_to_it; sequence(NULL, near_it?, open); ],
        at_it?,
        eat);
]
```

The rough English version of this code is "make food the target object, and until you are `at_it`, do the following: `go_to_it`, and when you are `near_it?`, open your mouth. Then, when `at_it?`, `eat`."

The sequence syntax looks like a function call in a C-like language, but here it is more akin to a macro. The macro defines a structure which activates the first argument until the second becomes active; then it activates the third argument. The value of the `sequence(...)` expression is an object reference of a sort; here it points to a particular unit which will be a component of the final unit network.

The syntax for making food the target object is more interesting. The `:` syntax denotes using the name on the left hand side as a scope for evaluating the code on the right hand side; it is similar to `with` in Pascal. So this means within `attributes`, within `target`, activate `food`. `attributes` is the visual attribute-representing agency, and `target` is a working memory copy of that agency. Activating `food` in the context of `target` has the effect of putting the `target` memory into the same state that `attributes` would be in if food were perceived. Rather than thinking of this as an assignment statement which is executed and then done, instead it is best to think of it as a condition which is stimulated while this behavior is running. (However, working memory state persists until it is changed.)

The value of the expression `attributes: target: food` is an object reference; the reference carries with it the associated context. Both elements in the brackets `[ ]` are thus of the same semantic type; the `[ ]` operator returns a new object which merely activates all the enclosed objects, simultaneously.

`go_to_it` and `eat` are simple:

```
go_to_it = sequence(NULL, attributes: target, forward);

eat = sequence(open, food_in_mouth, close);
```

The only thing worth noting here is that `attributes: target` denotes a unit which is active when the `target` memory contents match the state of the `attributes` agency.

Lets look next at part of the `attributes` agency:

```
attributes<target, grab_attributes, source, destination, ...>:
{
    ...

    shape:      { square; round; flat; };
    size:       { small; medium; large };
    color:      { red; green; blue };
    distance:   { at; near; medium; far };

    block       = k-line(shape: square, size: small);
    box         = k-line(shape: square, size: large);
    wall        = k-line(shape: flat, size: large);
    food        = k-line(shape: round, size: small, color: green);

    at_it?      = and(target, distance: at);
    near_it?    = and(target, distance: near);

    ...
};
```

The { } syntax serves to enclose multiple declarations in the same scope. Making declarations in the scope of `attributes` defines that agency's contents. The names in angle brackets define working memories for the agency: essentially copies of the agency, but with a nested scope. Within `attributes`, there are nested agencies: `shape`, `size`, `color`, and `distance`. These represent the independent components of what is currently perceived. Within `shape`, we see simply a list of three names. A name introduced as a declaration without an associated definition simply creates a structureless unit to go with the name.

The activation levels of `square`, `round`, `flat`, `small`, etc. define the current state of `attributes`. The memories are declared as K-lines. In this world, blocks are things which are small and square, so the `block` K-line puts `attributes` into a `small` and `square` state. `k-line` is a macro, like `sequence`, that builds a characteristic unit structure. In this case the structure built activates the indicated units when the K-line unit is active, and vice-versa. (There are a number of possible unit microstructures that can serve here; potentially activators and recognizers can be distinct units, though identical at the specification level. Alternatively, a single K-line unit can serve; the feedback implied can be damped, so activation decays over time without stimulus.)

`at_it?` and `near_it?` are here defined as simple recognizers, units which are active when both specified condition are met. Again, `target` here is considered to be active when the `target` memory state matches the `attributes` state.

Now lets look at something more complicated, the behavior that sorts blocks into boxes with matching color:

```
fill_box =
[
    attributes: source: block;
    scripts: destination: find_matching_box;
    trans_frame;
];

scripts<source, destination>:
{
    find_matching_box      = attributes: target: [ box; color: grab_attributes ];

    default_source         = attributes: target: source;
    default_destination    = attributes: target: destination: target;
};

get_it = sequence(go_to_it, at_it?, < attributes: grab_attributes: ^*, grab >);

deliver = sequence(go_to_it, at_it?, drop);
```

```
trans_frame =
[
    0.5 * (scripts: [ source: default_source; destination: default_destination ]);

    sequence(
        [ scripts: source; get_it; ];
        carry;
        [ scripts: destination; deliver ];
    );
]
```

`fill_box` makes a couple of working memory assignments to set up context, and executes `trans_frame`, which actually does the getting and putting. `trans_frame` weakly loads two default scripts, then gets an object and delivers it to a location. The syntax for weak activation is simply multiplication of an object reference by a number. In this case this has the effect of making the script assignments defaults. When the `trans_frame` activates `scripts: source`, there is no contention, and `default_source`, which was loaded into `source`, is activated. This then copies `attributes: source` into `attributes: target`, and the `get_it` behavior then gets the target object. In this case the source object, set up by `fill_box`, is a block. When it comes time to deliver the block, then the default `trans_frame` destination script is overridden: instead of `default_destination`, `find_matching_box` is activated, since its 1.0 activation exceeds the 0.5 activation of `default_destination`. `find_matching_box` takes the steps necessary to ensure that only a box matching the color of the block picked up will do.

There are three other pieces of new syntax here. `^` means enclosing scope, and `*` means "currently active units in this scope". So attributes: `grab_attributes: ^*` means set the `grab_attributes` memory to be the same as the current `attributes` state. The angle brackets, in this context, specify an ordered sequence of activations (as opposed to `[ ]`, which indicates concurrent activation). This is a kind of built-in macro for a structure that activates each argument until it is active, then stops activating it, and proceeds with the rest of the sequence. It is necessary here because the creature is storing away attributes of the object it sees, but once the object is grabbed, it will no longer see it. The color of the object is then used to match the destination block, as above.

Other Representations

The language described above is really just a fancy syntax for specifying a kind of connectionist network, with a built-in K-line-like memory model. The network could just as well be programmed by graphically editing a view of the network. That is, in fact, the most natural way to observe a creature's cognitive operation as it is being simulated, and I will provide at least the ability to view the network in this form. Ideally this view will be manipulable as well, allowing new behaviors to be built by literally wiring together existing ones.

4 The World

The inspiration for my simulated world is A. K. Dewdney's book "The Planiverse" [9]. It describes a fictional two-dimensional world, inhabited by sentient beings. Unlike Abbott's Flatland [1], in the Planiverse there is gravity – the planet ("Arde") the creatures ("Ardeans") live on is a circle; they live on the surface. Thus they move back and forth along a line. A Planiverse creature is shown in Figure 7. (All Planiverse images copyright 2001 A. K. Dewdney. Used by permission.)

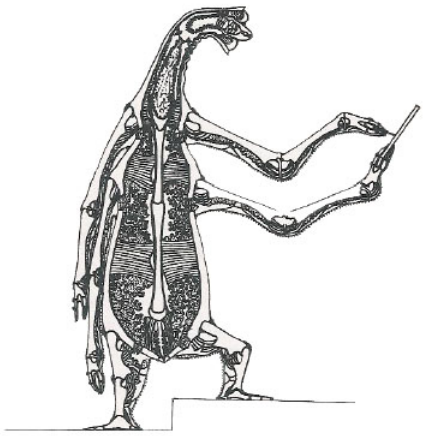


Figure 7: An Ardean.

Having only a single dimension of movement is not very interesting, but an extra dimension is provided by underground construction. A multi-level house is shown in Figure 8. Hinged stairways with pull-chains allow the Ardeans to ascend, descend, or walk across a lower level.

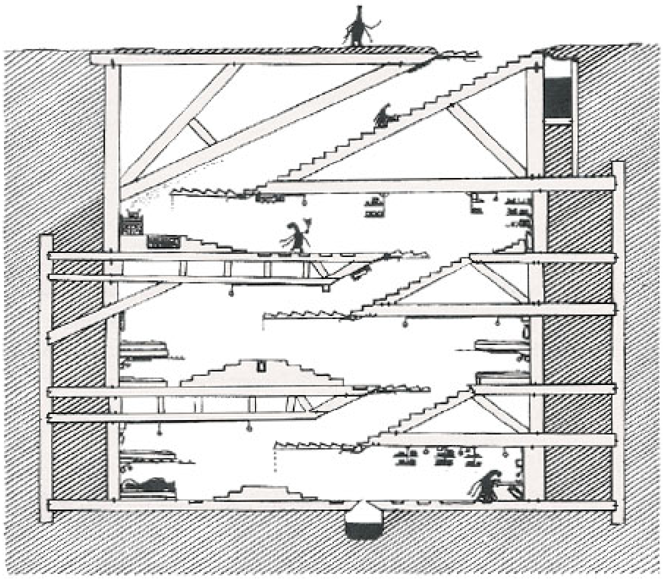


Figure 8: An Ardean house.

Many interesting mechanical devices are possible in the Planiverse [7, 8]. Springs, wedges, hinges, levers, and cables all work as expected. Gears of a sort are possible, but they are tricky to use, because they can't have ordinary axles. Even complex machines are possible. A two-dimensional steam engine is shown in Figure 9.

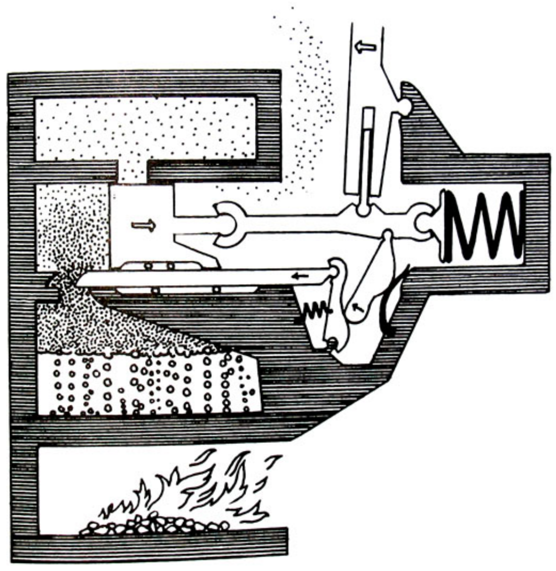


Figure 9: A two-dimensional steam engine.

My simulated world will be similar to the Planiverse world. The creature anatomy used for Dewdney's Ardeans is my target anatomy to simulate. I plan to have underground houses with hinged stairways, that the creatures must navigate, and objects the creatures can pick up and manipulate. These objects will include balls, food, furniture, toy blocks, boxes, and possibly game pieces.

However, my simulator will be limited to rigid body physics – the steam engine is beyond the scope of this project. Likewise, Ardean boats would be interesting, but would require modeling fluids. If possible I would like to implement ropelike objects, however. Ropes have novel uses in the Planiverse: a bag is simply a rope held at both ends.

5 The Creatures

The creature behaviors will be built up incrementally, in terms of simpler existing behaviors. This methodology is similar to a subsumption architecture [4], but in a classic subsumption architecture, behaviors are pre-wired, and there is no K-line-based memory substrate. Here the working memory model effectively allows the network to be rewired at runtime, and the learning processes I plan to implement will also change the network as time evolves.

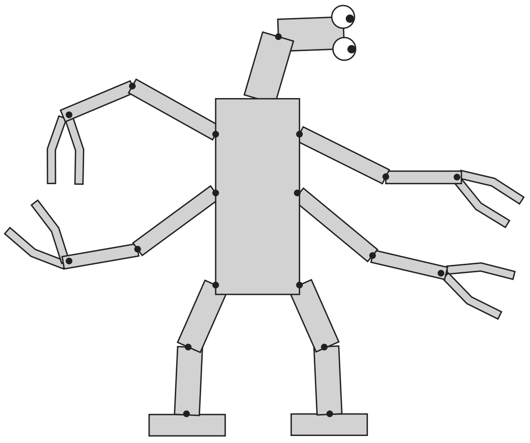


Figure 10: A possible creature body.

I propose implementing the following scale of increasingly complex behaviors:

1. Build a simulated robot body corresponding to an Ardean. The creatures themselves, viewed as mechanical systems, are fairly simple. They require fewer degrees of freedom than comparable three-dimensional beings. Having two arms on each side of the body allows for complex object manipulations, requiring only a two-fingered hand on each arm. Here again the constraints of the two-dimensional world make the system design simpler. A sample body is shown in Figure 10.

2. Program simple behaviors to make the creature walk. Following the subsumption paradigm, walking will rely on lower-level balancing behaviors.

3. Program behaviors to visually identify, pick up, and put down objects These behaviors are analogs of the simple ones described in Section 3, but here the physical environment is more complex. Expressing the behaviors as simple programs will be a challenge. It is possible that some low-level behaviors will need to be evolved rather than programmed. This is acceptable, as long as the higher-level behaviors can be programmed.

4. Construct an emotional architecture that dictates goal states and priorities of behaviors. The emotional architecture will use SOM concepts such as cross-exclusion, censors, and suppressors to arbitrate between competing desires [15, Section 16].

5. Add a low-level architecture to the simulator for creating new long-term memory records of agency activation states. This will fundamentally change the creature's mental landscape, and set the stage for learning.

6. Implicit knowledge formation will result from traditional machine learning techniques applied to the K-line memory model [16], and

7. Explicit knowledge formation will result from learning behaviors running in the creature. There are many proposed learning behaviors in SOM for me to test. These include mechanisms for building new hierarchies of recognizers [15, Section 10.9], and learning by *uniframing*, or finding general descriptions that unify previously separate pieces of knowledge [22], [15, Section 12.3]. I will also explore learning using Piagetian schemas [10].

   Finally, with all of this infrastructure in place, I will be ready to work on the (seemingly) most complex behaviors seen in humans:

8. Problem-solving, and

9. Language.

   Problem-solving approaches to try include trial and error, trying to break goals down into simpler subgoals, reformulation [15, Chapter 14], and debugging almost-right plans [20]. Examples of problem-solving tasks are complex object manipulations, such as figuring out how to fit an oddly-shaped object into a box; and tasks requiring planning, such as navigation.

   Language opens the door to social interaction, and a large new class of problems to solve. The ideas in SOM on language are built on the frame concept, which my memory model is designed to support. The physical model for language will involve atomic transmission of phoneme-like units to nearby creatures. No attempt will be made to make the creatures use English; I will develop a communication language appropriate to the creatures' environment and concerns. However, the language will be of a form such that a human user of the system can communicate with a creature.

6 The Cognitive and Physics Simulators

Much of the value of this work will lie in others being able to use the language. To do so effectively, they will need access to both the cognitive and physics simulators, so I will need to make both of them available collectively as a freely available runtime system. The functionality required of the cognitive simulator is sufficiently specific that no off-the-shelf software is likely to suffice, but it might reasonably be supposed that an off-the-shelf physics simulator would be appropriate. Unfortunately, there does not appear to be such a simulator that is (1) 2d, (2) freely available, (3) software-controllable, and (4) robust and reliable. Therefore, I will need to write both simulators myself.

There is a common conceptual thread between the cognitive simulator component of this proposal and the physics simulator. Both are numerical simulations of complex processes that are too difficult to solve analytically or symbolically. In the physics domain, this is a common problem; numerical solutions to problems are often necessary. In the cognitive domain, there has been much work in connectionist architectures, but the memory abstractions that are the basis of my architecture, and the overall guiding principles of SOM, offer hope for novel results.

The Cognitive Simulator

There is nothing especially technically difficult about the cognitive simulator. It will simulate a network of basic units, with numerical activation levels. At each time step the activation levels will be updated according to units' inputs, and to sensory information. The connections in the network can also change as new K-lines are formed. Even working memory assignments will be implemented as instantaneous rewirings, representing the lower-level memory operations discussed in Section 2.

The units themselves can have a variety of output functions, including threshold, sigmoid, and simple sum. All are useful for building different kinds of circuits. The units need no internal state beyond an activation level, and need no explicit time-dependence; each time a unit's value is recalculated, it simply updates based on its inputs. Decay of activation over time can be achieved with self-input. The granularity of the simulation time will be such that reaction to events in the physical simulator must occur in a small number cognitive simulator steps, further enforcing biological-style programming.

The Physics Simulator

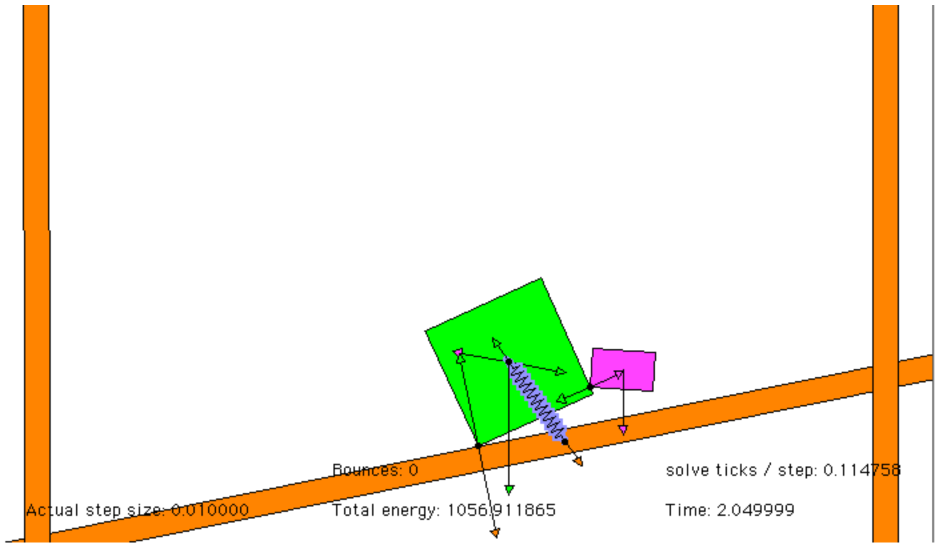
A rigid-body simulator works by numerically integrating the equations of motion describing a system, and resolving object contact events to modify the equations [6]. Any of a number of standard integration techniques may be used, e.g., Runge-Kutta, or Bülirsch-Stoer [17]. To simulate creatures that can walk and manipulate objects, articulated bodies are required. This causes the equations of motion to depend on forces generated at the joints; these forces must be solved for at each time step by further numerical methods. This involves solving linear systems of equations, by e.g. LU decomposition, or conjugate gradients. Other forces are simple functions of state (e.g. gravity, forces due to springs).

I use the equations of motion in Newtonian form. This is the simplest approach, but it means that every constraint in the system causes there to be a redundant coordinate in one of the objects constrained. The redundant coordinates can drift due to accumulated roundoff error, causing constrained objects to separate. Therefore, stabilization techniques are required to correct the drift; a number of methods are available to do this (e.g., [5]). There are fewer, and simpler, equations needed to integrate in 2d vs. 3d.

The most complicated part of the simulation is dealing with objects in contact. When two objects come into contact, an impulsive force must be generated to cause the objects to bounce or stop relative to each other. The impulsive forces may be solved for in a way similar to solving for the ordinary forces, with a correction for friction during the bounce. If the objects do stop against each other, a new constraint must be created in the system to represent the continuing force keeping them from interpenetrating. This changes the equations needed to integrate; the integration then proceeds from the new state. Similarly, when objects in contact separate, constraints are removed, and integration proceeds from the new state. Objects remaining in contact must slide against each other, with friction. Detecting and handling the contact and separation events correctly is an active area of research, and I believe I have some new ideas to contribute to the literature here.

The physics simulator must also interact with the cognitive simulator. At each time step, sensory data based on object positions is transferred to sensor unit states, and effector unit states are transferred to joint torques. Vision is handled by forming a one-dimensional image as viewed from each eye object in the system, and transferring the contents to the appropriate unit states.

A screenshot from a prototype simulator is shown in Figure 11.



```
Bounces: 0                        solve ticks / step: 0.114758
Actual step size: 0.010000    Total energy: 1056.911865    Time: 2.049999
```

Figure 11: A prototype two-dimensional rigid-body simulator.

References

[1] Edwin Abbott Abbott. *Flatland: A Romance of Many Dimensions.* 1880.

[2] P. Agre and D. Chapman. Pengi: an implementation of a theory of activity. In *The Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Seattle, 1987. American Association for Artificial Intelligence, Kaufmann.

[3] William Bechtel and Adele Abrahamsen. *Connectionism and the Mind.* Blackwell Publishers, 1991.

[4] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[5] Michael Bradley Cline. Rigid body simulation with contact and constraints. Master's thesis, The University of British Columbia, July 2002.

[6] Murilo G. Coutinho. *Dynamic Simulations of Multibody Systems.* Springer-Verlag, New York, 2001.

[7] A. K. Dewdney. Two-dimensional science and technology. London, Ontario, Canada, 1980.

[8] A. K. Dewdney, editor. *A Symposium on Two-Dimensional Science and Technology*, London, Ontario, Canada, May 1981.

[9] A. K. Dewdney. *The Planiverse: Computer Contact with a Two-Dimensional World.* Simon and Schuster, 1984.

[10] Gary L. Drescher. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence.* The MIT Press, 1991.

[11] Kenneth Haase. Framer: A persistent, portable, representation library. In *Proceedings of the European Conference on AI*, Amsterdam, 1994.

[12] Robert A. Hearn. Building grounded abstractions for artificial intelligence programming. Master's thesis, Massachusetts Institute of Technology, May 2001.

[13] Marvin Minsky. A framework for representing knowledge. In Winston [23].

[14] Marvin Minsky. K-lines: a theory of memory. *Cognitive Science*, 4(2), 1980.

[15] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1986.

[16] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[17] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.

[18] Luc Steels and Rodney Brooks, editors. *The Artificial Life Route to Artificial Intelligence: Building Embodied, Situated Agents.* Lawrence Erlbaum Assoc., 1995.

[19] Lucy A. Suchman. *Plans and Situated Action.* Cambridge University Press, New York, 1987.

[20] Gerald Jay Sussman. *A Computer Model of Skill Acquisition.* Elsevier Science, 1975.

[21] Michael David Travers. *Programming with Agents: New metaphors for thinking about computation.* PhD thesis, Massachusetts Institute of Technology, June 1996.

[22] P. H. Winston. Learning structural descriptions by examples. In *Psychology of Computer Vision* [23].

[23] P. H. Winston, editor. *Psychology of Computer Vision.* McGraw Hill, 1975.