Recursive Interfaces for Reactive Objects

Michael Travers

MIT Media Laboratory
20 Ames Street
Cambridge, MA 02139
(617) 253-0608
`mt@media.mit.edu`

ABSTRACT

LiveWorld is a graphical environment designed to support research into programming with active objects. It offers novice users a world of manipulable objects, with graphical objects and elements of the programs that make them move integrated into a single framework. LiveWorld is designed to support a style of programming based on rule-like agents that allow objects to be responsive to their environment. In order to make this style of programming accessible to novices, computational objects such as behavioral rules need to be just as concrete and accessible as the graphic objects. LiveWorld fills this need by using a novel object system, Framer, in which the usual structures of an object-oriented system (classes, objects, and slots) are replaced with a single one, the *frame*, that has a simple and intuitive graphic representation.

This unification enables the construction of an interface that achieves elegance, simplicity and power. Allowing graphic objects and internal computational objects to be manipulated through an integrated interface can provide a conceptual scaffolding for novices to enter into programming.

KEYWORDS: programming environments, objects, direct manipulation, visual object-oriented programming, agents, rules

INTRODUCTION

The Vivarium Project was begun in 1986 by Ann Marion and Alan Kay [8] with the goal of exploring new, biologically inspired directions for novice programming environments. Inspired by the task of simulating animal behavior, we also wanted to make our computational environment more life-like. In part, this meant that objects should not be mere passive respondents to the commands of an external user or program, but have autnomous behaviors of their own. They should react to the user and to each other. The computational models used in these designs were based on *agents* in Minsky's [11] sense: small modules that define simple behaviors, but can be combined into large systems that exhibit complex behavior; and that execute autonomously, yet can interact with and influence one another.

The Vivarium project has generated some earlier software environments: Agar [13] and Playground [4]. In these systems the computational realization of an agent was something of a cross between a rule and a process. These previous efforts concentrated on simulating the environment, the specification of agents for the animate objects, and the interaction between the creatures and their environment. LiveWorld is an effort to improve upon these systems in several ways: first, by providing a flexible and powerful object system and interface, and second by incorporating a more developed notion of a computational agent, along with a theory of how such agents may be created and debugged. This paper deals only with the first of these goals.

The motivating simulation domain for these systems was animal behavior, but the larger goal was always to find fruitful new models for programming in general. Many other systems may be described in terms of objects with autonomous but interacting behaviors; i.e., video games, MUDs, and virtual worlds in general. I've coined the term *animate system* to refer to this class of environment consisting of multiple reactive objects.

Creating animate systems requires manipulating many different kinds of objects (worlds, actors, agents, and sensors, for instance). These must be accessible, manipulable, extensible, and combinable by the user. Yet we don't want to overwhelm novices with complexity. LiveWorld's solution is to use a novel object system, Framer [6], which permits objects to be viewed and manipulated at varying levels of detail. Complex objects may be treated as black boxes by beginning users, while more advanced users can open them up to see and change the parts inside. To accomplish this, all levels of parts are made out of the same basic structure, the *frame*.

The design values of LiveWorld include:

- **tangibility** – objects should be concrete, accessible to the senses, and capable of direct manipulation.

- **reactivity** – objects should be aware and capable of reacting to each other. The programs that drive objects should be running all the time, so that user interaction can trigger object behavior.

• **improvisation** – it should be easy to explore new possibilities by making incremental modifications to existing objects.

• **learnability** – the system should provide "conceptual scaffolding" that allows the learning of new skills to be rooted in existing skills.

RECURSIVE OBJECT STRUCTURE

LiveWorld's object system and interface are tightly integrated and must be considered together. The interface design was inspired in part by Boxer [3], which pioneered the use of recursive containment as a basis for a general computational medium. LiveWorld borrows the powerful idea of recursive containment from Boxer, and integrates it with object-oriented programming. The object system is based on Framer, a knowledge representation tool with prototype-based inheritance, developed by Ken Haase at the MIT Media Laboratory. Framer's hierarchical object representation meshes perfectly with a Boxer-like graphic interface.

Object system

Framer provides a single structure, the frame, out of which more complicated structures are built. A frame has a name and a location within a structure that is similar to a hierarchical file system. All frames except the root have a container or *home* frame and may also have contained frames or *annotations*. Frames also have an optional value (which may be any Lisp object, including another frame) and an optional prototype (which must be a frame).

The graphic counterpart to the frame is the *box*. From the user's perspective these two constructs are identical, i.e., it is possible for "the user to act as if the representation were the thing itself" [7, p97] for all components of the system. LiveWorld represents Framer's hierarchical structure by means of spatial containment.

Figure 1 illustrates a simple LiveWorld construction. Frame names are indicated in boldface, so **x** is an annotation of **my-turtle**, which is in turn an annotation of **cast**. Frame values follow the name, so the value of the **x** annotation is 24. Boxes may be open or closed, with

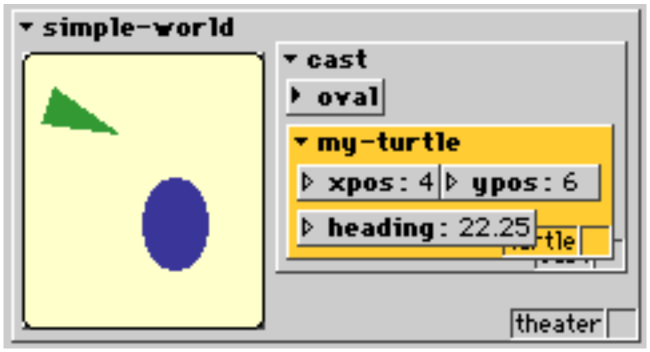


Figure 1: A simple world with two graphic objects. Note that theater, stage, cast, objects, and slots are all realized from the basic frame/box.

this state being controlled by a handle in the shape of a triangle (styled after the hierarchical folder display in the Macintosh Finder [1]). A closed box displays only its name and value, while an open box will display the contained annotations, the prototype (in a recessed box in the lower-right) and a resize handle.

Recursive annotation

A frame can thus serve as an object in the usual sense (by containing other frames that define its properties and behaviors) or a slot (by virtue of the fact that it can have a value). Frames may also be used as collections of objects, and because any frame may serve as a prototype, they also take the place of classes in a traditional class-based OOP system. Boxes can also serve as buttons and other interface objects by giving them a special click handling method (see figure 3).

This is quite different from traditional object systems, where classes, slots and objects are very different sorts of entities. This unification permits LiveWorld to have a simple and uniform interface.

In addition to providing a simplification of traditional constructs, frames make it easy to build advanced programming constructs that are difficult or impossible to realize in conventional object systems. These include facets, demons, dependency networks, and histories. It also makes it easy to use frames to represent information *about* frames. For instance, display information for frame boxes is stored in (normally invisible) frame annotations to each displayed frame (**%box-position**, for instance). If you display these frames, they get their own display annotations, recur–sively.

It should be noted that LiveWorld and Framer are recursive only in the sense that the basic frame structure is defined recursively, and most functions that deal with it call themselves recursively to work their way up or down the frame hierarchy. It does not imply that a frame can contain itself.

Prototype-based

Prototype-based object languages [10, 14] are an alternative to the class-based object schemes used in more traditional OOP languages. In a class-based object system, every object is an instance of a class. Although classes are usually represented by objects, they are of a distinctly different order than normal objects. Sometimes the set of class objects and related descriptors are called meta-objects. In contrast, a prototype-based system has no classes or special class objects. There is no rigid distinction between classes and instances or between objects and meta-objects. Instead of defining objects by membership in a class, they are defined as variations on a prototype. Any object may serve as a prototype for other objects.

The advantages of prototype-based programming are simplicity and concreteness. It eliminates a whole set of objects (class descriptors) from the environment, and simplifies the specification of inherited properties. In a sense it is specification by example. For example, the properties of elephants might be defined by a prototypical elephant (Clyde) who has color **gray** and mass **heavy**. Properties of Clyde become defaults for the spinoffs, so all elephants will be gray and heavy unless their default is overridden, as in the case of pink elephants. With prototypes, the specification of defaults and exceptions are done in exactly the same way, that is, simply by specifying concrete properties of objects.

There are some indications that mental representation of categories makes use of prototypes [9]. Whether this translates into any cognitive advantage for prototype-based computer languages is open to question. What prototypes do accomplish for programming is to smooth out the development process, by easing the transition from experimentation to system-building.

Theatrical metaphor

Graphic objects are implemented by special frames called *actors* that appear in *theaters*. Theaters offer two views of their objects, a *cast* view and a *stage* view. Both the cast and stage are frames themselves, and the actor frames are actually contained within the cast frame and draw themselves in the stage frame. The two views of the objects are interconstrained (so that, for instance, dragging an object will continuously update the relevant slot displays).

The system provides a graphics library of basic actors. These include shapes and text-actors, pictures, and turtles. The library itself is a theater and new objects are created by using the standard cloning commands, so there is no need for specialized palette objects.

The link between graphic and non-graphic objects may help novices who are experienced users of direct manipulation interfaces make the transition to programming, both by showing them a different view of their actions and by permitting them to interact with computational objects in a familiar way.

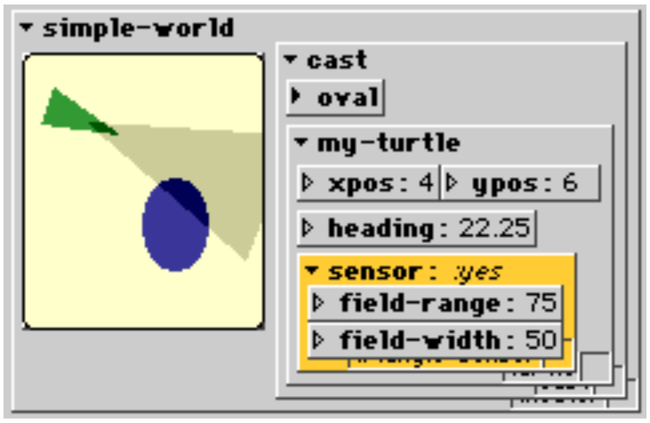


Figure 2: **simple-world** after dropping a sensor into the turtle and opening it. The sensor displays as a translucent range overlay; and recomputes its value as the objects move.

Sensors

In LiveWorld, sensors are frames within an actor that have a value (i.e., :yes if the sensor is activated, or a number for a numerically-valued sensor). Sensors also have many properties associated with them – such as their range and what kind of objects they are sensitive to. Because frames can have both a value and annotations, this is easy to represent. The sensor in Figure 2 contains internal slot frames (such as **field-range**) that allow it to be customized. It may be seen as a slot (since it has a value) or an object with slots of its own.

INTERFACE DYNAMICS

Dragging

Most operations in LiveWorld are accomplished by moving objects around (dragging). There are several forms of dragging; the most basic kind changing only the position of the object, with no semantic effects. Lift-and-drop dragging removes the object from its container and allows it to be placed in another. A lifted object actually lives in a special frame (**limbo**) and the distinction is indicated to the user by a drop shadow. Clone-and-drop dragging actually creates a new spinoff object which is then treated as if it had been lifted. Clone-and-drop is the most common way to create new frames.

One interesting feature of LiveWorld is immediate redisplay of inherited values. For instance, if a graphic object is cloned, and then the original is resized, both the original and the clone will be updated interactively. This is a powerful illustration of inheritance and may have pedagogical value. On the other hand, it can cause surprises for users who aren't aware that values are being inherited.

Selection and Commands

LiveWorld operates on the select-and-operate user interface model. The selected frame can be deleted, its value or prototype changed, etc. The Messages menu allows the user to send messages to the selected object, as well as serving as documentation for available operations. The Slots menu shows what slots are defined for that object and selecting an item from that menu will create a version of that slot frame local to the selected object.

Cloning

Typically, new objects in LiveWorld are created by copying existing objects via the clone-and-drop interface and modifying selected properties. The new object has the object it was cloned from as its prototype, and (in general) inherits all the properties of the original. Graphic objects may be cloned in either their box representation or their graphic representation.

By default, cloning is shallow. This means that if a frame with contained frame structure is cloned, only the top-level frame is copied. For most internal frames (i.e. those used as slots) this is adequate, since the slot values in the original will be accessible in the new one. But in some cases, internal objects must be copied. For instance, in Figure 2, if a clone of **turtle** is made, the slots **xpos**, **ypos**, and **heading** don't need to be copied (and aren't), but **sensor** must and will be, since the new turtle ought to have eyes of its own. Frames specify for themselves if they need to be copied when their container is copied. The frames that need to be copied are generally those that specify local sub-parts like sensors, or generate side-effects like animas (described below).

After a frame is cloned it must be installed (dropped) somewhere. Frames can have a **fits-within** slot that specifies what sort of object the frame may be dropped into; the interface will only allow the object to be dropped appropriately (i.e., sensors can only be dropped into actors). Frames can have **install** methods which allows them to specify side-effects that happen as a result of being dropped.

Inheritance
One of LiveWorld's strengths is illustrating the nature of inheritance. Graphic objects that inherit properties from a prototype will update dynamically if the prototype is changed. Inherited values in boxes are shown in italics; they too update dynamically if changed.

Framer only offers single inheritance, but because slots are objects in their own right it is often possible to simulate the effects of inheriting from multiple objects. For instance, if we have a turtle (which can inherit only from the prototypical turtle) and we want to add behavior that defines a mass and forces acting upon it, we can copy in the appropriate slots from the prototypical physical object (i.e., **mass**, **gravity-agent**). Effectively, the turtle can inherit properties from several objects.

PROGRAMMING

LiveWorld defines a message-passing protocol over Framer objects. *Actions* (LiveWorld's term for methods) are themselves frames, so they can be copied from object to object and otherwise manipulated.

Actions contain code written in Lisp extended with primitives for Framer and for message-passing (see Fig. 4 for an example). Future versions will also support actions programmed in other languages (such as Logo or a natural-language-like scripting language). It's relatively easy to support multiple languages by making language interpretation an object property. In Lisp, messages are sent to objects through the **ask** primitive. The Ask menu lists the available messages for the selected object and serves both as documentation and interface.

Agents are specialized actions that arrange to be run repeatedly by virtue of containing an *anima* object (see below). A typical agent consists of a conditional that tests a sensor and conditionally takes an action (by sending a message). A variety of schemes exist to permit agents to control each other; these are beyond the scope of this paper. Further development will allow the specification of networks of agents, handling conflicts between agents, and support for agents with explicit goals.

LiveWorld provides a library of actions and agents. Beginning users can create animated creatures by cloning objects out of this library. The objects also provide a starting point and templates for more advanced users who are ready to start programming. The transitions between these stages (copying, modifying, and finally creating agents from scratch) are relatively smooth, with each stage providing scaffolding for learning the next.

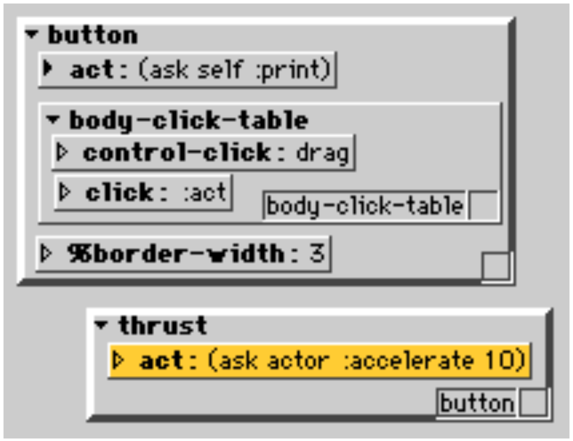


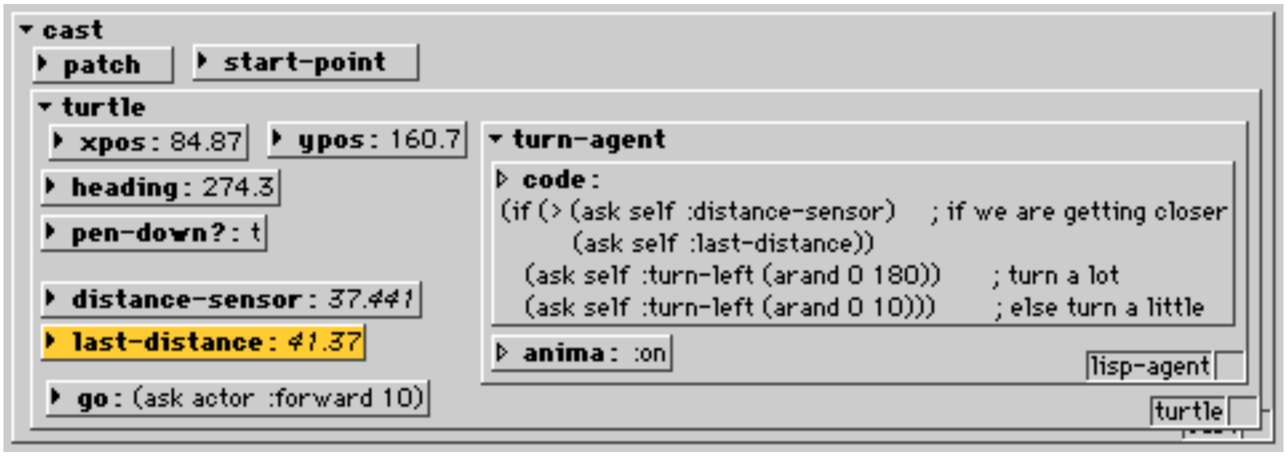
Figure 3: Customizing a frame in order to define buttons.

Redefining the interface through itself
The interface that LiveWorld presents is partially definable in LiveWorld itself. This is done by means of click tables that define mouse actions. A mouse gesture is really defined by two parameters: the part of the box clicked upon, and the particular gesture (modifier keys) held down. Arbitrarily, we pick the first of these to be the most important and define click-tables for various components (including one for clicks on the graphic view of an actor). Click-tables contain entries for particular clicks (i.e. **option-shift-click**) whose values are messages to be sent to the frame that contains the table.

Click tables inherit like anything else, so this technique may be used to define specialized objects such as buttons (see Figure 3). Here, an action object is specialized to have a different appearance (via the **%border-width** slot) and a special action to take on an ordinary mouse click (it will send itself a :act message). The **button** object can then be cloned and the clones given code of their own to define their action, as is done with the **thrust** button.

Reactivity and concurrency
Standard computer languages and interfaces are based on a strictly top-down, command-driven model of execution. This model is inadequate to build simulations of systems in which multiple objects may be acting independently. Worse, it teaches students to think of control in a top-down style that may hinder understanding of systems in which control is distributed, that is to say almost any real-world system.

The image of LiveWorld is that there are many objects, many agents, and many actions going on at once. Objects are not controlled through a central locus, but determine their own actions. Ideally, every part of the system should



```
▾ cast
  ▸ patch   ▸ start-point
  ▾ turtle
    ▸ xpos: 84.87   ▸ ypos: 160.7
    ▸ heading: 274.3
    ▸ pen-down?: t
    ▸ distance-sensor: 37.441
    ▸ last-distance: 41.37
    ▸ go: (ask actor :forward 10)
    ▾ turn-agent
      ▸ code:
      (if (> (ask self :distance-sensor)    ; if we are getting closer
             (ask self :last-distance))
        (ask self :turn-left (arand 0 180))      ; turn a lot
        (ask self :turn-left (arand 0 10)))      ; else turn a little
      ▸ anima: :on
                                                  lisp-agent
                                                  turtle
```

Figure 4: The actors for the klinokinesis example. The turtle has two agents: one makes it go forward at a constant rate, the other makes it turn a random amount that depends on the food concentration gradient. The value of the **last-distance** slot is also computed by an internal agent (not shown).

be constantly aware, monitoring its conditions and reacting to it.

LiveWorld supports concurrency by enabling objects to be called in the background. The interface to this capability is through objects called *animas,* which repeatedly send a message to their containing frame. Animas may be turned on and off individually, or *en masse*. Inheritance may be used to control a group of animas.

Animas illustrate a classic tradeoff in object-oriented design. Animas are add-in objects which enable any containing object to become an agent or process. The alternative would be to make animacy a property of the containing agent itself. The advantage of this second approach is simplicity, since it has one less object type. The anima solution, however, has the advantages of modularity in that a specific ability is encapsulated in a specific kind of object, and that object can be included anywhere); and of providing an interface or affordance to the user for adding or removing animacy. LiveWorld's hierarchical object system permits a compromise between these solutions. Since animas can be hidden in higher-level objects and automatically inherited, the user can ignore the fact of their existence at first, and discover them later when ready to "open the hood" of an agent.

Animas are normally running in the background, and it is common for the user to be making changes to the world or to code while other activities are going on. For instance, in the klinokinesis example below, you can move the food patch that the turtle is tracking and watch it adapt. This serves to lower the barriers between running and coding.

In the present implementation, animas are called in a round-robin fashion. This is adequate for most examples, but does not really qualify as true pseudo-parallelism. A facility to support pseudo-parallelism by performing all side-effects in parallel has been constructed and is undergoing evaluation. Again, the recursive annotation ability of framer helps, since the system can store intermediate values locally in annotations.

FINAL EXAMPLE

In Figures 4 and 5, a simulation of klinokinesis, a navigation technique used by very simple animals to navigate. The strategy is to move forward (via the **go** agent) and to change directions based upon gradient differences from a food source. The rule is: if the concentration is increasing, turn a small amount, otherwise, turn in a random direction). The liveness of the simulation allows you to interact with it (for instance, by moving the food patch while the turtle is running, or by changing the direction manually by using the **turn-right** button, inserted from figure 3).

PROBLEMS AND FUTURE WORK

LiveWorld's unified representation has some negative consequences. For instance, LiveWorld builds a variety of

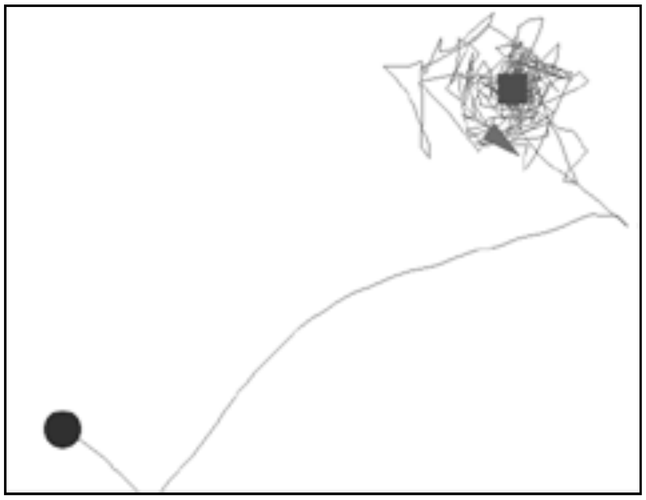


Figure 5: The stage for the cast in figure 4. The turtle started from the point at the lower-left and found its way by "smell" to the food patch at the upper right

constructs using the basic containment relationship. As a result, from the user's perspective containment is overloaded and its various uses somewhat *ad hoc*. The relation between a sensor inside a creature is not the same as that of an anima inside an agent. Since the graphic representations are the same, this may be confusing to the user.

In general, there is a tension between simple, powerful, and general constructs and the interface goal of making specific structures for specific tasks. Take object palettes in a traditional program: because they are special objects, they may easily be given special properties such as floating or a special appearance. LiveWorld, though, has been carefully designed so that no special palette objects are necessary! We have achieved simplicity, but at the cost of being able to easily specialize the design of specific elements.

There are many unresolved issues revolving around the deletion and modification of prototypes. Framer by default does not allow the deletion of an object with spinoffs. The interface encourages loose patterns of cloning, but this can cause problems as dependencies are created which make later modification difficult. I have experimented with a modified delete function that permits deleting an object with spinoffs, by copying the slots of the prototype into the spinoffs. This preserves the characteristic of the spinoffs but loses type information.

User testing of the system is planned; some issues to be investigated are the learnability of the object system; whether the availability of all objects in graphic form is empowering or confusing, and whether the particular forms of conceptual scaffolding actually provide greater learnability.

RELATED WORK

Boxer

Boxer also uses graphic containment of boxes to support a variety of computational constructs. However, Boxer and LiveWorld have very different notions of what a box is. In Boxer, boxes are understood in terms of an extension of text editing. A Boxer box is like a character in a text string that can have internal structure, which will be other lines of text and boxes. Boxes have no fixed spatial position, but do have a position based on textual ordering. There are no classes of boxes (except a few built-in system types) and no inheritance.

In contrast, LiveWorld boxes are full-fledged objects, and the interface presents them as such. Boxes are dragged and copied with the mouse rather than with keyboard editing commands. Where the interface of Boxer is rooted in text editing (it is in fact based on the Emacs text editor), the interface of LiveWorld is rooted in object-oriented graphics. Both systems extend their root metaphor by allowing for recursive containment.

LiveWorld's advantages over Boxer is primarily in implementing inheritance and a more modern interface (mouse-based rather than text-based) with better integration of graphics. The advantages of Boxer is that its textual metaphor makes it natural to impose an order over the items in a box, and may be more accessible to some users with a textual orientation.

Self

Self [14] is currently the prototypical prototype-based object-oriented programming system. LiveWorld's (really Framer's) major contribution over and above what Self provides is in making slots be first-class objects and thus enabling recursive annotation and containment. Another difference is that in Self, an object generally does not inherit values from its prototype, but from a separate object (the parent). While this has certain advantages, LiveWorld's method is simpler and, I hope, easier to convey to the naive user.

Self has an exploratory graphic interface [2] which, while having some commonality with that of LiveWorld, is exploring essentially different issues. For instance, both systems emphasize animation, but use it in very different ways. The Self interface animates its object representations (akin to LiveWorld's boxes) in a fashion that makes them seem real to the user, but this animation is outside the realm of the user's control. LiveWorld is more concerned with allowing the user to control the animation of graphic objects, leaving the more abstract, structural representation of objects unanimated.

Rehearsal World and ARK

Other systems which LiveWorld may be compared with include Rehearsal World [5] and the Alternate Reality Kit (ARK) [12]. LiveWorld borrows its theatrical metaphor from Rehearsal World, which used it in more elaborate form (i.e. referring to message-passing as "cues"). The purpose and feel of the two systems also have many points of commonality: both aim to provide a friendly and dynamic world where programming and direct manipulation are integrated. The main differences are LiveWorld's unusual object system and its emphasis on concurrent reactive objects. Rehearsal World's programs, in contrast, are generally "button-driven", that is, driven directly by user actions.

ARK is another environment with goals similar to that of LiveWorld. In this case the commonality is in providing a lively, reactive feel, and in integrating graphic objects with computational objects. Other shared properties include prototypes (ARK's prototypes work more like those of Self), step methods to drive concurrent objects, and a special limbo state (called MetaReality in ARK) for objects in transition.

Both ARK and Rehearsal World, as well as many other visual environments, rely on a separation of user levels. Components are built by sophisticated users, generally outside of the visual environment, while more naive users are allowed to connect these components together graphically but not to open them up or modify them. LiveWorld, on the other hand, tries hard to make every object openable and accessible.

CONCLUSIONS

LiveWorld combines a novel form of object-oriented programming with a direct manipulation interface to create an environment that's concrete, reactive, and flexible. To date, the system has been used chiefly as a tool for exploring the space of possible agent-based programming styles.

The fact that all objects are presented graphically by default, that is, without the cognitive overhead of having to request their display in an inspector, is a subtle yet powerful change in the way that the programmer relates to objects. In LiveWorld, every object and every slot is an affordance for action as well as a prototype for improvised modifications.

ACKNOWLEDGMENTS

Ken Haase and Alan Ruttenberg developed tools that made this work possible. Alan Kay and Ann Marion began the Vivarium project and have provided support through Apple Computer. Mitsubishi Electric has also provided support. Amy Bruckman, Andy diSessa, and Linda Hershenson commented on drafts of this paper.

REFERENCES

1. Apple Computer. *Apple Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, Reading, Massachusetts, 1987.

2. Chang, B.-W. and Ungar, D. Animation: From Cartoons to the User Interface. In *Proceedings of UIST '93*. (November 3-5, Atlanta, Georgia), 1993, pp. 45-56.

3. diSessa, A.A. and Abelson, H. Boxer: A Reconstructible Computational Medium. *Communications of the ACM 29*, 9, 1986, 859-868.

4. Fenton, J. and Beck, K. Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. In *Proceedings of OOPSLA '89*. 1989, pp. 123-137.

5. Finzer, W. and Gould, L. Programming by Rehearsal. Xerox Palo Alto Research Center Research Report SCL-84-1, 1984.

6. Haase, K., Framer. 1992, unpublished software:

7. Hutchins, E.L., Hollan, J.D., and Norman, D.A., Direct Manipulation Interfaces, in *User Centered System Design,* D.A. Norman and S.W. Draper, Editor. 1986, Lawrence Erlbaum Associates: Hillsdale NJ.

8. Kay, A. Vivarium Papers. Unpublished essays, 1990.

9. Lakoff, G. *Women, Fire, and Dangerous Things*. University of Chicago Press, Chicago, 1987.

10. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of First ACM Conference on Object Oriented Programming Systems, Languages & Application*. Portland), 1986, pp. .

11. Minsky, M. *Society of Mind*. Simon & Schuster, New York, 1987.

12. Smith, R. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *Proceedings of SIGCHI+GI'87: Human Factors in Computing Systems*. Toronto), 1987, pp. 61-67.

13. Travers, M., Animal Construction Kits, in *Artificial Life: SFI Series in the Sciences of Complexity,* C. Langton, Editor. 1988, Addison-Wesley: p. 421-442.

14. Ungar, D. and Smith, R.B. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*. (October 4-8. 1987, Orlando, Florida), 1987, pp. 227-241.