Experiencing SELF Objects: An Object-Based Artificial Reality*

BAY-WEI CHANG (bay@self.stanford.edu)
DAVID UNGAR† (ungar@self.stanford.edu)
Computer Systems Laboratory, Stanford University, Stanford, California 94305

1 Introduction

Programming is hard. Programming forces the programmer to deal with many things at once, to grasp complex relationships between elements in a program, and to manage the many dependencies between those elements. Programming taxes the programmer's short term memory.

The SELF project strives to make the programmer's job easier by combining good language design, efficient implementation, and a user interface tightly coupled to the language. SELF is a dynamically-typed, prototype-based, object-oriented language. This paper describes the approach we have taken with the design of the user interface for SELF. Detailed description of the language and implementation can be found in [1, 2, 6, 11].

Our prototype user interface for SELF provides browsing and inspecting of SELF objects by combining an object-based model with an artificial reality. Emphasizing the problem-domain objects rather than views of those objects discards a layer of indirection found in conventional window-based user interfaces. Placing those objects in an artificial reality reduces the cognitive load on the programmer, by exploiting the programmer's already internalized knowledge of how objects in the physical world behave. Our goal is to make the user interface invisible, and thus make SELF objects and the SELF world *real*.

*This work has been generously supported by a National Science Foundation Graduate Fellowship and National Science Foundation Presidential Young Investigator Grant #CCR-8657631, and by Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

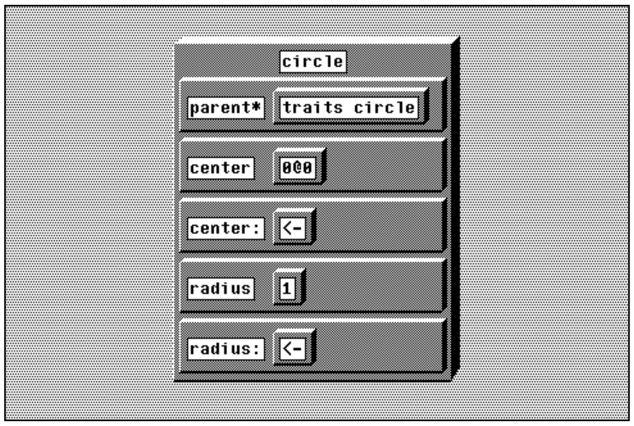

Figure 1. A SELF object.

The next section provides an introduction to the prototype SELF user interface via a brief tour of some of its aspects. Section 3 discusses the power of an artificial reality in a programming setting and how adoption of an artificial reality in the SELF interface works for the programmer. Section 4 examines the benefits of an object-based model and section 5 follows with a look at the personality of SELF objects. Finally, we conclude with some observations of the lessons to be learned from this work.

2 The SELF artificial reality

The SELF artificial reality consists of SELF objects populating otherwise empty space. Figure 1 shows a typical SELF object.

This object, a box with other boxes glued on its face, can be grabbed with the mouse, which can be thought of as a "virtual" hand, and moved about in its world (the screen). The object keeps up with the programmer as it is dragged around, floating in front of any other objects in the world.

The object in Figure 1 represents the prototypical circle in SELF, with center at the origin and unit radius. Like all SELF objects, this circle object consists of a set of slots, which are the wide boxes on the face of the object. Each slot has a name—the words on its face—and contents. The contents of a slot, which is just another SELF object, is represented by a box on the face of the slot, indicating containment. The circle object of Figure 1 has five slots: two slots hold the center point and the radius of the circle, two slots are assignment slots that permit the contents of the center and radius slots to be changed, and the final slot is a parent slot (indicated by the asterisk after the name of the slot) that holds an object that the circle inherits from.

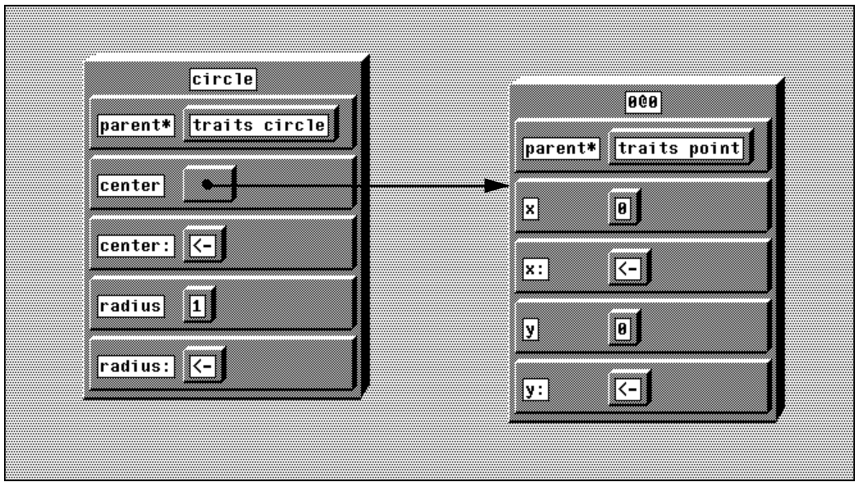

Figure 2. Objects are connected by arrows.

Some SELF objects have names.[1] Well-known objects, like the circle prototype object, have a name that is the path of slots from the root object (known as the "lobby"). Thus, sending the message `circle` to the lobby returns the prototypical circle object. Other objects, like integers and points, know how to print themselves and are named accordingly. Since the current SELF user interface is used as a browser and inspector, most objects encountered are well-known and thus are named. (The bulk of objects in an executing program have no name, since they are created dynamically and typically cannot be reached by any path starting at the lobby.) The name of an object appears at the top of the object. The name of an object that is contained in a slot is imprinted on the "contents" box on the slot face.

To examine the contents of the circle's `center` slot, the programmer grabs that contents box (by clicking on it with the mouse). The contents of the slot, the point object `0@0`, appears on the screen ("sprouts"), connected by an arrow to the contents box from which it sprouted. In the future, sprouting will cause the contents box to detach and smoothly grow into the full object; a contents box will be just like a condensed form of an object.

Sprouting is simply a special case of grabbing; after sprouting an object, the programmer can move it to wherever he wishes before releasing the mouse button. As he moves it, any connecting arrows stretch like rubber bands. The movement of

[1]Names in SELF are not part of the object insofar as the programmer does not bestow a name to each object. The language semantics have no separate provision for objects to be given a name; the SELF artificial reality infers the names it uses.

the objects and arrows occurs at a high enough frame rate[2] to provide the illusion of reality.

In studying the workings of objects in a program, the programmer will typically sprout many objects on the screen. Objects that are no longer being examined can be removed from the screen (currently, by clicking a different mouse button). The resulting world is a number of objects arrayed on the screen, with intrinsic relations being shown by arrows connecting objects and extrinsic information often being shown by the objects' placement on the screen relative to one another.

Figure 3 shows part of a simple shape hierarchy. The prototypical circle inherits from the circle traits object, which holds shared behavior for all circles (in a similar way that a Circle class in a class-based language would hold behavior for all instances of circles). The circle traits object in turn inherits from the shape traits object, which is also a parent of the triangle traits object.

Although both circle traits and triangle traits have a slot containing the shape traits object, only one shape traits object is on the screen. Since there is only one shape traits object in the system, only one such object exists in the SELF artificial reality. If the programmer sprouts the parent slot of triangle traits when the shape traits object is already on the screen**,** an arrow connects the slot to the existing shape traits object, and the virtual hand (the mouse cursor) moves to that object.The integer object zero in Figure 3 provides another example. It is referenced by both the x and y slots of the `0@0` point object as well as many other places in the system. This is one way that the SELF artificial reality maintains the identity of objects.

Both the SELF artificial reality and the objects in that world have certain personality traits that contribute to the programmer's interaction with the system. A closer examination of these traits follows in the next three sections.

3 Artificial reality in the SELF interface

The user interface is the means with which the programmer accesses and manipulates the objects in his program. Traditional interfaces present themselves as a tool, a framework for the programmer's interactions with the objects in the system. The interface is clearly visible; there is never any doubt that the programmer is interacting with the interface, which translates his actions to access the problem-domain objects on his behalf. The interface is a barrier between the programmer and the objects in the system.

Applying the artificial reality paradigm to the user interface helps to minimize that barrier. Objects in an artificial reality behave according to the laws of that artificial world. When those laws are similar to the physical world, the user is able to use his intuition to understand an object's behavior. In ARK [9, 10], for example, objects have mass and velocity, and obey such physical laws as gravity and inertia.

[2]30 frames per second for medium-sized objects on a GX-equipped Sun Sparcstation-1.

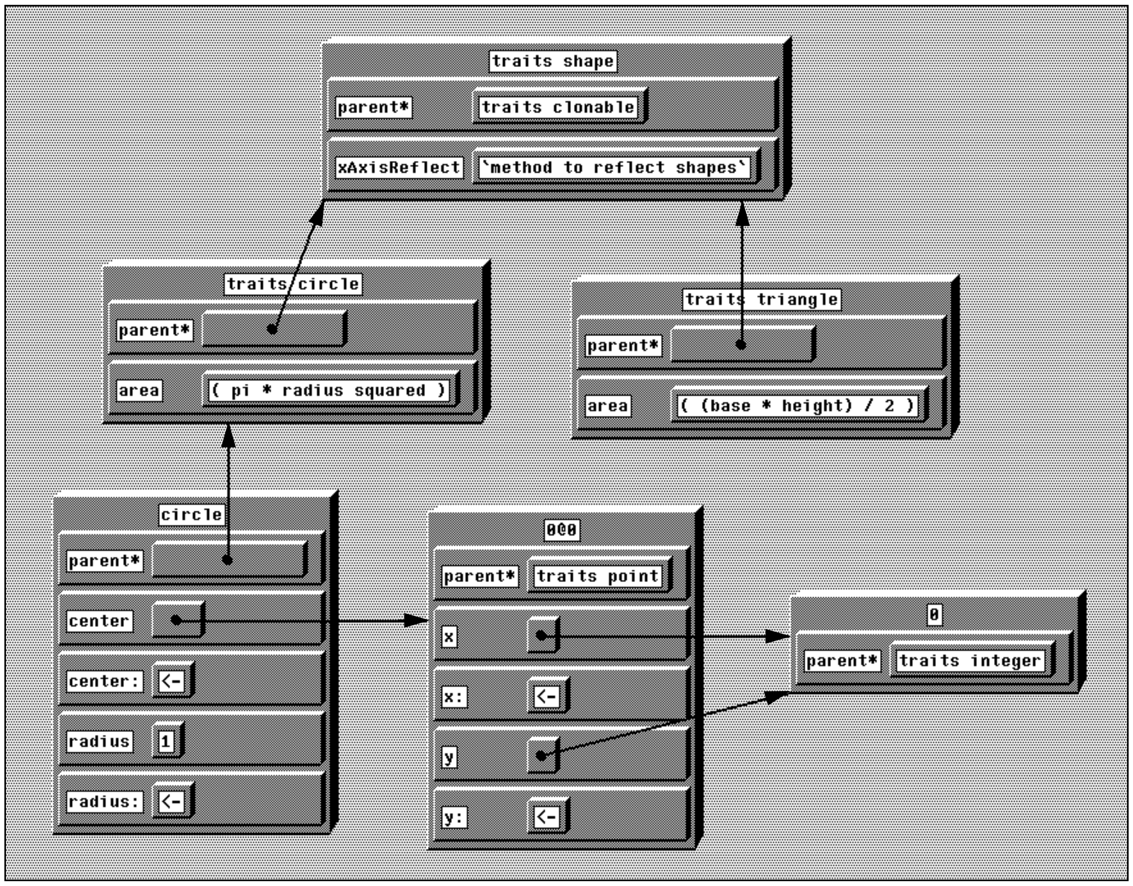

Figure 3. Multiple slots may point to the same object.

Tossing an ARK object causes it to continue moving in its original direction indefinitely. If there is friction in the ARK world, the object will slow down and come to a stop. Behavior like this is easy for the user to understand, since it is the sort of behavior of objects in the everyday world with which the user has been interacting with since birth. Once the user accepts that the artificial world on the screen behaves in large part like the real world, he will come to forget that he is dealing with a user interface. New tasks in the environment are learned more quickly and naturally.

But artificial reality is powerful not only because it provides an environment in which behavior is easy to learn and understand, but also because it removes a layer of indirection between the user and the objects in the system, thus eliminating much conceptual overhead needed to operate within the interface. This is important in a programming environment, in which the concern is less for the learnability for a novice user than it is for the usability for an experienced programmer. The programmer need not spend as much of his attention on operating the user interface, since (in an artificial reality based on a physical world metaphor) those skills have already been internalized as part of his lifelong interaction with the physical world. The ideal is for the notion of an *interface* to vanish completely, to be replaced by a *portal* through which the user enters an artificial world populated by the objects at hand.

The personality of an artificial reality is determined by the laws that the world and the objects in the world obey. The SELF artificial reality relies on a physical world metaphor. Some of the aspects of the personality of the SELF artificial reality are:

- *Space*. Initially, the world on the screen contains but a single object, typically the lobby object. From the lobby, the programmer can navigate to any well-known object in the system, simply by sprouting objects in slots and following those paths. As objects are sprouted, they are placed by the programmer throughout the space of the world.
- *Solidity*. The three-dimensional look of boxes, illuminated by an off-screen source of light, make objects solid. Moving an object moves the actual object around the screen, not an outline, which contributes to maintaining the solidity of objects. Achieving a high frame rate further reinforces solidity by avoiding discontinuous jumps.
- *Physical interaction*. The programmer manipulates objects by grabbing them to pick them up. Sprouting a slot is also performed by grabbing the contents box on the slot. Since the objects are in an artificial world, the mouse-controlled cursor is thought of as an artificial hand in that space, an extension of the programmer's real world hand.
- *Limited depth*. Although the SELF artificial reality does not yet have a true third dimension, objects may overlap one another. Since objects themselves are three-dimensional, the effect is a world of limited depth in which objects float in front of and behind one another near the surface. Picking up an object brings it the closest to the programmer; as a result, there are also no collisions between objects, since objects being moved about are in front of all other objects. We are investigating ways of using true depth, shadows, and collisions in the SELF artificial reality.

The physical metaphor in the SELF artificial reality permits the programmer to think about and work with SELF objects without the additional weight of dealing with an obtrusive interface. By concentrating on maintaining the illusion of an artificial reality, it attempts to make the interface itself invisible.

4 Object-based interfaces

Conventional interfaces treat problem-domain objects as objects hidden within the system, accessible only via the tools of the user interface. These tools provide *views* of the problem-domain objects, each tool showing some aspect of objects in the system. Examples are the Smalltalk-80 [3] and Trellis [7] environments. Smalltalk-80 uses different tools for different activities: browsers show classes and the methods defined in classes, inspectors show the instance variables of objects. This style of user interface can be called tool-based or activity-based [4].

The activity-based model emphasizes the manipulation of user interface tools to provide views of problem-domain objects. In contrast, the object-based model treats problem-domain objects as the objects to be manipulated in the user interface. Problem-domain objects are made concrete in the interface, and the programmer identifies objects with their user interface representation. No such identification occurs in activity-oriented interfaces, because a tool is used to view many different objects over its lifetime. That the tools themselves are concrete is of little use to the programmer, since the tools are not the problem-domain objects that the programmer is concerned with.

Identifying problem-domain objects with their representation in the interface affects the programmer's mental model of the objects in the system. If the identification is complete enough, the programmer will come to think of the objects depicted on the screen as the actual objects they represent; their behavior in the interface will then contribute to the programmer's mental model of the objects in the language. Therefore it is critical that the behavior of objects in the interface closely follow the semantics of the language. Behavior inconsistent with the language semantics will inevitably confuse the programmer; conversely, a careful design supporting the language model can make the interface a powerful tool in aiding the programmer in understanding and working in the language.

Examples of object-based interfaces include the Star interface [8] and the Macintosh Finder [12]. The Macintosh Finder represents directories as folders. Opening a folder yields a window, which may contain folders as well as icons representing files. Folders and file icons are directly manipulatable in the interface; they may be moved about in the window, or moved to another folder. Since the user identifies files with their icons, moving an icon from one folder to another means the same to the user as moving the file from one directory to another, but in place of some abstract notion of a directory there is the concrete image of folders containing files.

However, the object-based model breaks down in some places in the Finder. A window representing an open folder does not share full identification with the folder; it is really another view of the folder. Not only do both the folder window and the folder icon exist on the screen in different places at the same time, a window of an open folder cannot be moved to another folder like a folder icon can. The object-based model in the Finder applies only to icons.

The Mjølner environment [4] also adopts an object-based model, in which elements of programs are represented by hierarchical windows. The hierarchical windows in Mjølner are well-matched with the block structure of the languages it is designed to support, clearly expressing containment relations. In addition, it is an explicit goal of the Mjølner environment to create an identification between the elements of the program and individual windows; for example, referencing an element that is already somewhere on the screen does not result in a new window on that object; instead, the programmer's attention is directed to the element already on the screen. The Mjølner environment is a big step in directing the programmer's attention away from the interface to the objects in his program. But we want to go farther.

5 The object-based model in the SELF interface

The SELF interface completely eschews the notion of windows, which imply a view into something, in favor of an artificial concrete reality. Using an object-based model in the context of an artificial reality allows the SELF interface to forge a much stronger identification of interface objects with problem-domain objects. It attempts to coax the programmer into thinking of an object in the interface as *being* the problem-domain object, rather than simply *representing* it. As a result, the programmer's mental model of the language will be more heavily influenced by the model presented in the interface. Since the interface makes the mental model explicit and concrete, the programmer should be able to think about and interact with objects more naturally.

Another consequence of integrating artificial reality and an object-based model is the elimination of a separate notion of tools in the user interface. Tools are indirect mechanisms for manipulating and examining objects; the object-based approach, coupled with an artificial reality to provide natural and intuitive interaction with objects, transfers those powers to the objects themselves. Objects in the interface should, through their very nature, supply all the functionality that the programmer will need of them: this is simply consistent with the concept of manipulating real objects in a real (well, artificial) world. Therefore the design of an interface shifts focus from the *functionality* of tools to the *personality* of objects.

Objects in the artificial world reflect their personality as objects in the SELF language:

- *Identity.* A SELF object has a definite identity independent of state changes. In the artificial world, it is a solid object with a single existence. Therefore, multiple references to that object will direct the programmer to one object on the screen, rather than creating multiple interface objects corresponding to the problem-domain object. Well-known SELF objects are globally accessible via a path from the lobby. An object on the screen reflects this characteristic by displaying this path like a name tag.

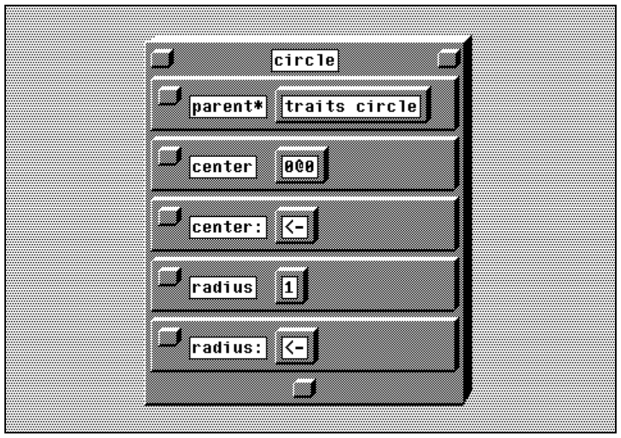

Figure 4. Object with reference buttons.

- *Composition*. A SELF object is composed of a set of named slots referring to other objects. In the artificial world, an object is a large box, with smaller boxes glued onto its face. Each smaller box has a name—the slot name—and yet another smaller box—the slot's contents—glued onto it.
- *Uniformity*. In the SELF language, everything is an object, down to the smallest integer. All things in the SELF artificial world also share a basic likeness; they look and feel the same.
- *Connectivity*. SELF programs are networks of interconnected objects. In the artificial world, relationships between objects are indicated by stretching arrows as links between those objects. Two objects are connected if one is in one of the other object's slots. The connections between objects can become as complex as the web of relationships demands, with many arrows pointing out from an object (each from a different slot) and many arrows pointing to an object (originating from slots in which the object is in). Since arrows stretch and remain connected to objects wherever they move, connections are maintained until the programmer explicitly decides to remove them from view.
- *Referral*. To understand an object's role in a program, it may be necessary to find all of the object's children, or all the objects that refer to it. To understand a slot's role in a program, it may be necessary to find all slots with the same name. An object in the artificial world can show the programmer a list of objects whose slots it is in, or a list of objects that inherit from it. A slot in the world can likewise show the programmer all objects containing its namesakes. Our current implementation of these functions uses buttons on the object and its slots. Figure 4 shows these buttons. In the future we hope to devise a more natural interaction with the object for referrals.

- *Interior vs. exterior.* Some information about objects can be considered part of the exterior of the object, while other information is part of the interior. SELF provides the notion of visibility of slots in its encapsulation scheme: public slots are accessible by anyone, private slots can only be accessed by the object and its descendants. The current SELF user interface provides access to the interior of objects, displaying all slots, public or private, for each object. Future work will define what it means to examine only the exterior of an object, its public slots.
- *Structure vs. behavior.* The structure of hierarchies of objects in SELF is provided by the inheritance of objects through parent slots. The SELF artificial world reflects this structural outlook in its box-and-arrows approach to inheritance. Behaviorally, however, the parents of an object are part of that object. In SELF this is as valid a model of the language as the structural model. We are studying ways to provide the behavioral model of objects as an alternative means of presenting inherited slots in the artificial world.
- *Intensional vs. extensional structure*. The extensional structure of objects considered as a system is determined by their relationships to one another by virtue of being in one another's slots. These relationships are made explicit by arrows which link objects when one object is sprouted from the slot of another. The positions of problem-domain objects on the screen have no semantic meaning; the programmer is the one who moves objects to their places. However, these positions, while devoid of information to the program, are usually full of information for the programmer; for example, positional information can indicate the inheritance relationships in a group of objects. The layout of objects on the screen are intensional.

We are investigating a technique we call *poses*, which acts like a snapshot, capturing the layout of objects on the screen into a pose object. Activating a pose object at a later time instructs all objects that were part of the pose to move to their former locations and all other objects to get out of the way, effectively reconstructing the layout that was captured earlier. Unlike rooms [5], the user does not move to recapture a certain layout; the objects in the world do. Also unlike rooms, objects maintain their identity and are never found in two places at once. A pose object does not include the actual objects; it is more like a specification, and when activated, a director. Poses promise to provide the convenience and functionality of maintaining several layouts of objects while still adhering to the artificial reality and object-based model of the user interface.

6 Conclusions

The SELF user interface encourages the programmer to think of objects in the artificial world as real. It attempts to strip away the conceptual overhead of a user interface, replacing tools with the objects themselves and the interface with an artificial reality. As a result, the programmer is brought closer to the SELF objects he is working with: rather than being restricted to viewing objects from afar and manipulating them indirectly, the programmer enters the SELF artificial world and can experience SELF objects directly.

The SELF interface described in this paper supports browsing the system and inspecting objects. Future work will continue to follow the principles laid out in this prototype to expand the interface into a complete programming environment. Among these principles are two observations we have found particularly important in an object-based artificial reality. First, the illusion of the artificial reality must be carefully maintained at all times. A lapse in the illusion, like a sunspot, disrupts communication more than its size would indicate. Even a small lapse can destroy the illusion, make the user consciously aware of the interface, and divert his attention from the problem at hand. Second, emphasis must be placed on the *personality* of objects in the system to reap the full benefits of an object-based interface. By discarding tools and views and giving responsibility to the objects themselves, the user interface places the emphasis on the problem-domain objects instead of interface-level objects, and the programmer can think about the objects rather than the tools.

7 Acknowledgments

Craig Chambers and Urs Hölzle contributed many ideas in discussions regarding the nature, functionality, and appearance of the SELF user interface. Randall B. Smith and his reality deserve the credit for setting our feet on the path of an object-based physical metaphor.

References

1. Chambers, C., and Ungar, D. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.

2. Chambers, C., Ungar, D., and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 49-70. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

3. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA (1984).

4. Hedin, G., and Magnusson, B. The Mjølner Environment: Direct Interaction with Abstractions. In *ECOOP '88 Proceedings*. Published as *Lecture Notes in Computer Science #322*, Springer-Verlag, New York, NY (1988) 41-54.

5. Henderson, A. D., and Card, S. K. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. In *ACM Transactions on Graphics*, 5, 3 (1986) 211-243.

6. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

7. O'Brien, P. D., Halbert, D. C., and Kilian, M. F. The Trellis Programming Environment. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 12 (1987) 91-102.

8. Smith, D. C., Irby, C., Kimball, R., Verplank, W., and Harslem, E. Designing the Star User Interface. In *Byte*, 7, 4 (1982) 242-282.

9. Smith, R. B. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proceedings of 1986 IEEE Computer Society Workshop on Visual Languages* (1986) 99-106.

10. Smith, R. B. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *Proceedings of the CHI+GI '87 Conference* (1987) 61-67.

11. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

12. Williams, G. The Apple Macintosh Computer. In *Byte*, 9, 2 (1984) 30-54.