The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility

Randall B. Smith, John Maloney†, and David Ungar

Sun Microsystems Laboratories

2550 Casey Ave. MTV29-116

Mountain View, CA 94043

randall.smith@sun.com, j.maloney@applelink.apple.com, david.ungar@sun.com

Abstract

Manipulating programs is hard, while manipulating objects in the physical world is often easy. Several attributes of the physical world help make it comprehensible and manipulable: concreteness, uniformity, and flexibility. The Self programming system attempts to apply these attributes to the world within the computer. The semantics of the language, the efficiency and fidelity of its implementation, and the architecture of its user interface conspire to make the experience of constructing programs in Self immediate and tangible. We describe the mechanisms used to achieve this goal, and illustrate those mechanisms within the context of an extended programming task.

I. Introduction

There is a satisfying directness in working with physical objects. You can pick things up, inspect, poke and prod them, or stick them together. Everywhere you go, it's pretty much the same thing — more physical objects. Move them out of the way, walk around them, poke them some more. It's all very straightforward, and in fact it makes programming systems seem pretty awkward and abstruse by comparison.

Of course it would be a difficult and dubious effort to literally replicate the physical world in the computer. But embodying key elements of our physical reality might bring benefits to the computing experience. The rise of object-oriented programming may be viewed as the outcome of an effort to allow computers to more directly model physical systems. However, even object-oriented programming systems don't really give you anything like that real-world sense of obvious straightforwardness, that sense of immediacy. We believe something more can be found by reexamining the essence of physical systems.

Three important characteristics of physical reality that are not truly present in most programming systems, even object-oriented ones, are concreteness, uniformity, and flexibility. In the Self system, we have tried to marry the cognitive advantages of objects with these three fundamental characteristics, thereby making programming less awkward, less abstruse. We hope that by so doing, not only will the expert become more productive, but

† Current address:
Advanced Technology Group
Apple Computer, Inc.
Cupertino, CA 95014

the novice will have an easier time scaling the learning slope.

We will describe specifically how aspects of the language, the user interface, and the implementation together move the system forward along these three dimensions of concreteness, uniformity, and flexibility. The Self-4.0 user interface (along with the elements of the programming environment it contains) is a part of the system that has not been described elsewhere, and will be our primary focus. But the language semantics and the implementation are equal partners, sharing the common design center.

Our exposition will visit each of the three elements in the Self mantra: concreteness, uniformity, and flexibility. Concreteness: The language is based on an object model that encourages a direct, copy-and-modify style of programming. The user interface independently adds concreteness by supporting immediate, direct access to any part of any application, including the environment, even while it is running. Uniformity: The language merges state and behavior, abolishes the class/non-class distinction, and uses object and message for everything. The user interface further increases the system's uniformity by using graphical objects down to the lowest levels, and by removing the distinction between run and edit. Flexibility: Combining concreteness and uniformity alone is a great aid to flexibility. The Self object manipulation facilities provided by the user interface, and the interface's reification of a central language level concept, the "slot," makes key language level manipulations readily available.

The implementation is an important enabler: by being faithful to the language semantics and by generating highly optimized code, the penalties one would expect to pay for semantic purity are greatly diminished. The programmer more often gets to think about his Self world of objects without concerns for implementation details, so will be less inclined to open "trap doors" into a more efficient language or to distort his programming style to gain efficiency.

The result is a system with a particular overall feel — offering liveliness, more immediate feedback, an almost tactile awareness of objects — a sense of the tangible reminiscent of the physical world.

II. Concreteness

Concreteness was one of the original motivations in the Self language design. A concrete object is easy to comprehend: it can be manipulated and directly inspected. The language's contribution to the sense of concreteness arises largely from its use of prototypes: the routine way to make new objects in Self is to copy and extend an existing object. One can see an example of what he will be working with, examine and even test it. In class-based systems, one instantiates from a description, and thus deals on a less concrete level.

The Self-4.0 user interface helps create a sense of concreteness and directness by making a Self window act like a world of tangible objects. It achieves this effect with four important characteristics: physical look and feel, unique Self level representation, reification of layout constraints, and composition through embedding. These characteristics arise from the default behavior of the basic graphical object, called a "morph." All display objects (circles, frames, buttons, pieces of text, and so on) inherit morph behavior, and are therefore kinds of morphs, acquiring concrete behavior by default. Also, any Self object can be viewed as a kind of morph called an "outliner," so the task of modifying or making new Self objects takes place in this concrete world. The four characteristics will be described one by one.

1] Physical look and feel

The physical look and feel arises from the way the user interacts with a morph. The Self interface is designed for a three button mouse. The left button is used to grab and carry objects across the screen. The grabbed object moves not as an outline of itself, but as a solid body, casting a drop shadow as though the user had lifted the object above the two dimensional world below. Because the Self user tends to work with many small morphs rather than a few large ones, the solid motion with drop shadow is

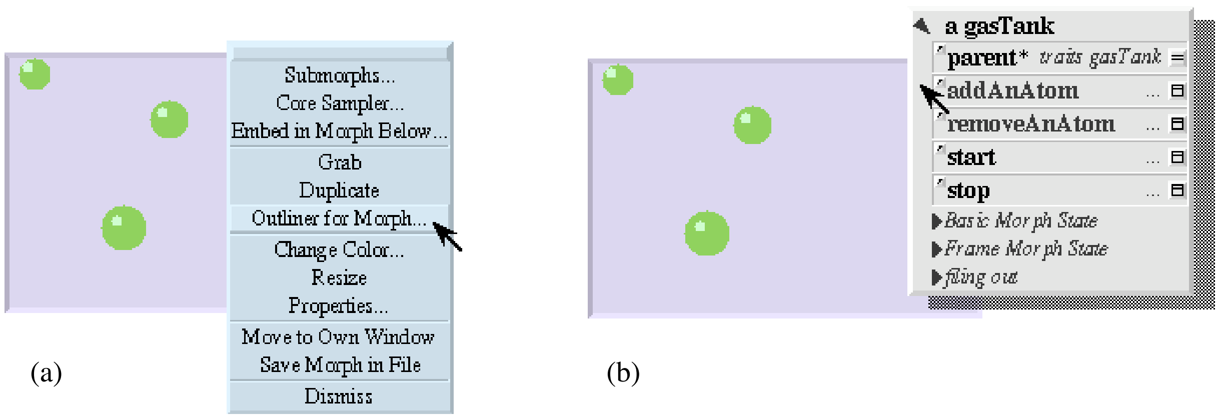


Figure 1. In a Self window, the user pops up the meta menu on the ideal gas simulation (a). Selecting “outliner” gives the Self-level representation to the user, which can be carried and placed as needed. (The italic items at the bottom of the outliner represent slot categories that may be expanded to view the slots. Unlike slots, categories have no language level semantics and are essentially a user interface convenience.)

an important part of making the system feel like a world of tangible objects.

The right mouse button is used to pop-up a “meta menu,” which is the same for all objects. The meta menu has items like “dismiss,” “change color,” “resize,” and “grab.” The grab item is useful for buttons and slider morphs, which override the default left button grab behavior in order to act like conventional user interface elements. The middle button is used by each object to pop-up a menu specialized to its own usage. A text editor, for example, might have cut, copy, and paste items on its middle button menu.

The use of pop-up menus might be called “action on contact.” Action on contact furthers the sense of direct, concrete feel. One well-known alternative to pop-up menus is the “selected object” paradigm, in which an object is first selected and then acted upon by graphically disjoint tools. (Consider most graphics editors or text editors for example, in which selected text or object can have various operations performed on it through pull down menus or palettes of tools.) While the selected object paradigm has its advantages (it’s good for single button mice, and presents a more visually manifest interface) it is fundamentally based on action at a distance. We chose the more physical action on contact approach to enhance the sense of concreteness. In another common alternative, the cursor becomes a tool for direct interaction. (For example, the cursor might become a paint bucket for doing object color fills in a graphics editor.) This is more direct than the selected object approach, but it still requires that the user slide the mouse over to a palette to change cursor modes. We believe the action on contact paradigm is both direct and quick, since the user specifies the object of action and invokes the menu in a single gesture: there is no need to acquire a distant target with the mouse. Action on contact is not achieved everywhere in the Self user interface (any use of a graphically disjoint tool violates the paradigm) but it is an important guiding principle that supports a sense of direct action.

To illustrate these points, we introduce what will become a running example throughout the paper. A Self-4.0 user (who might be a professional programmer, or a somewhat sophisticated and motivated end user) wishes to change and extend an ideal gas simulation. The simulation appears as a simple rectangular container (a “gas tank”) in which atoms, represented as disks, are bouncing around. (See Figure

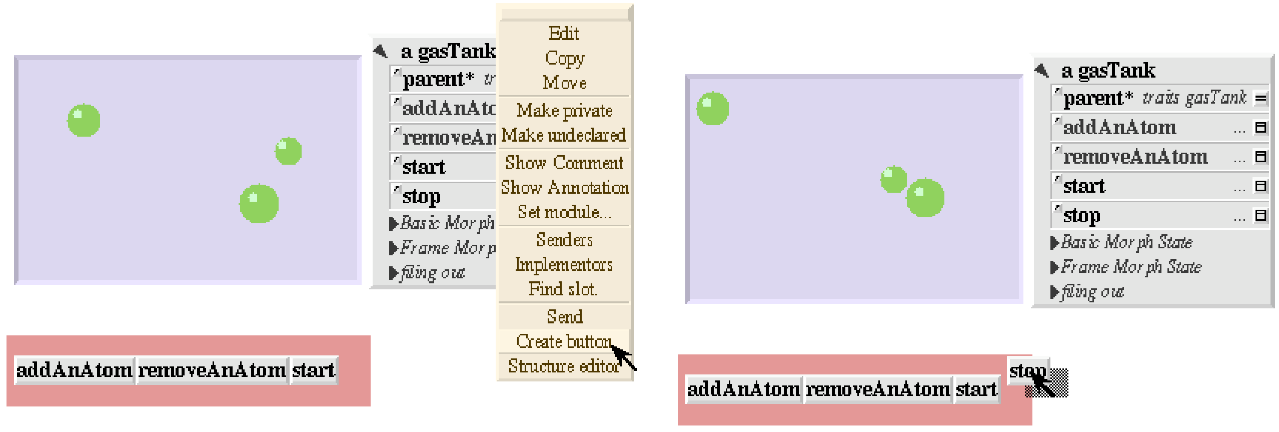
1). The user can go directly to the gas tank object as it is running, and pop-up the meta-menu to select “outliner,” which creates a Self object inspector, a language level representation of the gas tank.

In an outliner, we see the named slots that make up a Self object. In Self, everything is an object, and an object consists of named slots. Once the outliner is on the screen, the user can move it to a convenient location by grabbing with the left mouse button. The outliner allows the user to edit, add, copy, move, and remove slots by direct manipulation. The user can also open up a textual “evaluator” area on the outliner in which to type Self expressions, messages to be sent to the Self object. Thus, the outliner supports the language by providing ready, direct access to Self objects.

2] Single, unique Self-level representation.

The outliner is similar in some ways to the Smalltalk inspector, but one key difference helps create the sense of direct, concrete experience: only a single, unique outliner can be created per Self object. If the user pops up the meta-menu to ask for the outliner when the outliner is already in the world, the existing outliner slides into position near the user’s cursor. The use of animation rather than discontinuous jump supports the sense of concreteness, and provides extra information (where was the outliner being used?). The use of a single outliner per Self object supports the illusion that the outliner actually is the Self object: multiple representations of the same Self object would break down that illusion [4]. This restriction also helps decrease the screen clutter that many Smalltalk programmers are familiar with, in which several inspectors on the same object can become scattered or buried. Also, Smalltalk browsers and inspectors can become out of date giving obsolete and inconsistent views of data, whereas an outliner uses a background process to keep its display up to date. This furthers the sense of identification of the outliner with the Self object.

The outliner can be used to create a button that will send a message to activate a particular slot. For example, the user who is browsing the outliner for the ideal gas tank might notice a slot named “addAnAtom.” He can select “Create button” from a middle button pop-up menu on this slot. This creates a button bearing the label of the slot: “addAnAtom.” Clicking (left mouse button) on this button sends the message “addAnAtom” to the gas tank, making a new atom appear inside it. The slot-button correspondence takes advantage of the close match between the user interface level action (pressing a button on some display object) and the fundamental language level action (sending a message to trigger a slot in some Self object.)



(a) (b)

Figure 2. The middle mouse button pop-up menu on the “stop” slot (a) enables the user to create a button for immediate use in the user interface (b). This button will be embedded in a row morph, so that it lines up horizontally.

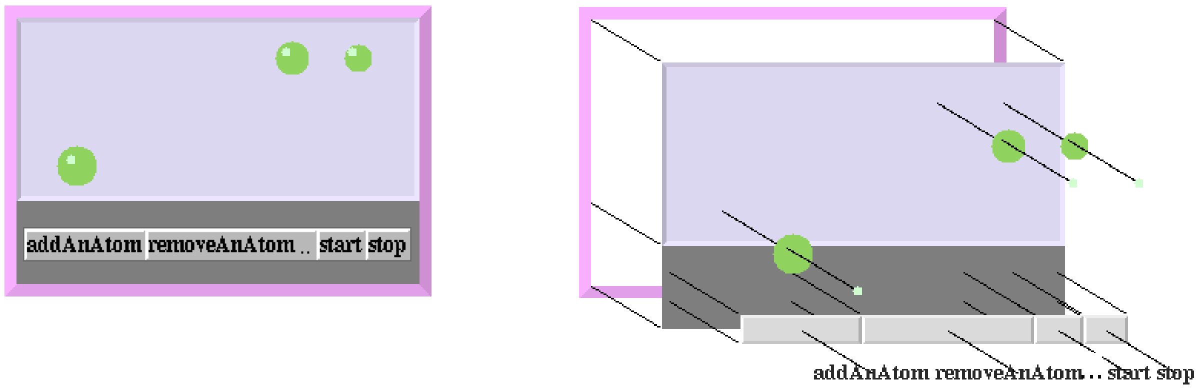


Figure 3. Composite graphical effects are achieved by embedding: any kind of morph can be embedded in any other kind of morph. The ideal gas simulation at left is a compound morph whose embedding relationships are shown at right.

3] Layout constraints are reified as morphs.

The user would like to make this and other buttons part of the new interface to the ideal gas simulation, somehow aligning them into a row. He can do this by creating a “row morph” from a system palette. (A system palette can be created from a middle button pop-up on the background.) The user carries a button over to the row morph, drops the button on top of the row, and selects “embed in morph below” from the button’s meta menu. The button then snaps into place on the row morph. The user repeats this operation for each of several buttons to create his row of buttons. (See Figure 2).

Because the row morph acts to hold the buttons horizontally aligned, we say that the row morph “reifies” this layout constraint. Layout reification contributes to concreteness and has immediate benefits: because the row is itself a kind of morph, it can be manipulated and accessed like any other morph. The row morph gives a single place to go to find out why the layout happens, or to change the way the layout constraint is maintained. For example, the user can get the row’s outliner and use its evaluator to send the row the message “topJustify,” causing the row to layout its submorphs with their tops aligned. The more ambitious user can program the row to layout its submorphs in some specialized way.

4] Compound graphical structure arises through “embedding” rather than “grouping.”

The user interface architecture also supports concreteness by allowing every morph to be embeddable, or to have any other morph embedded in it. Embedding buttons in a row is just one example. We call an embedded morph a “submorph” of its host. Complex morph-submorph hierarchical structures can be built up by embedding (outliners are at points 13 morphs deep). At this point in the ideal gas simulation, for example, the user might embed the row of buttons and the gas tank into a framed column morph. The column frame morph enforces the constraint that its submorphs line up one above the next. The resulting structure is illustrated in Figure 3.

An alternative to the physical embedding mechanism is the grouping metaphor as used in many structured graphics editors. Groups are normally invisible associations among objects, and therefore are less concrete than the first class graphical objects they contain. Embedding is based on a physical world attachment metaphor and lends a very tangible feel to working with morphs.

Summary

The sense of direct operation on tangible objects created by the four user interface aspects (physical look

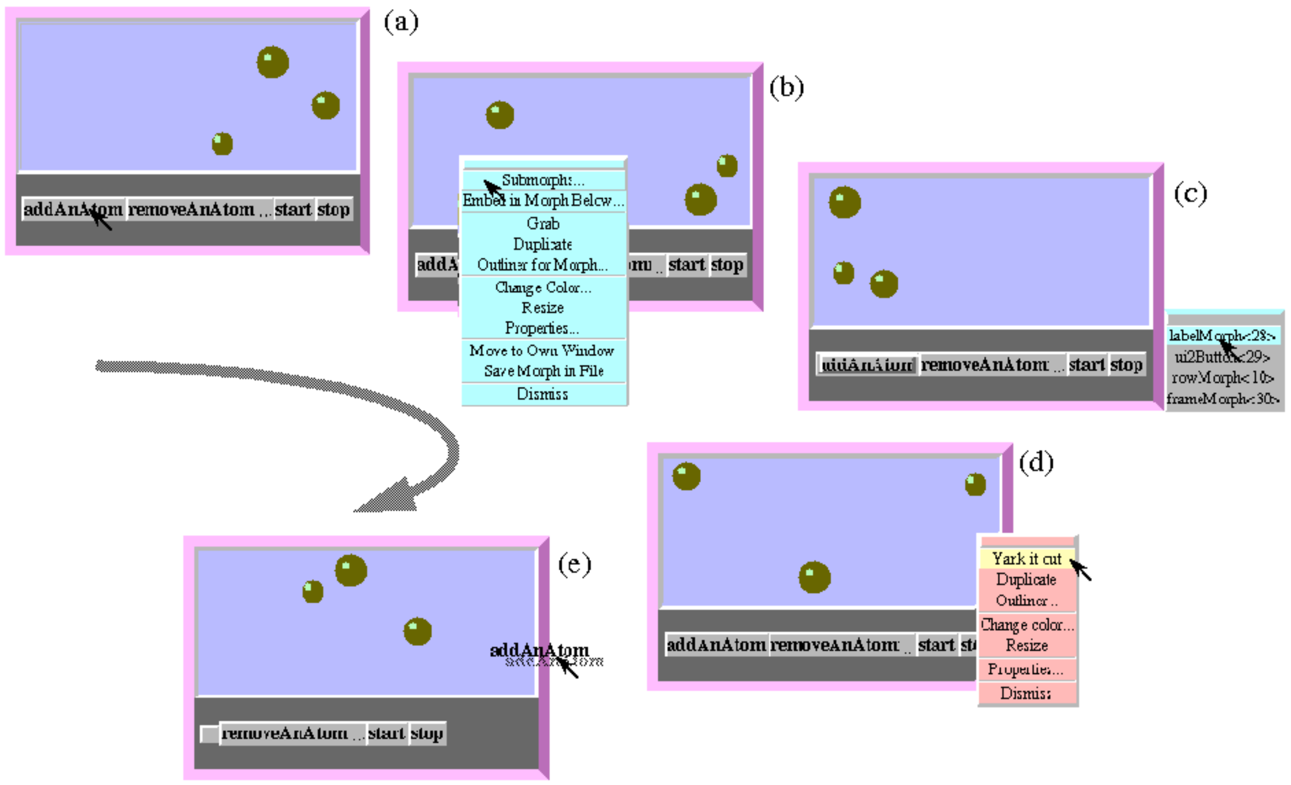


Figure 4. The user wishes to remove the label from the surface of a button. In this series of operations, the user starts by pointing to the label, selects “submorphs” from the meta menu, and selects the label from the resulting menu list. A menu of options is presented, from which the user selects “yank it out”. The button, which shrinks tightly around its submorphs, shrinks down to a minimum size when it has no submorphs.

and feel, single Self level representation, reification of layout, and embedding rather than grouping) adds to the sense of concreteness created by the language semantics. The language, being based on prototypes, creates an intellectual sense of concreteness by letting the programmer work with real data structures rather than descriptions, whereas the user interface is closer to motor-sensory perception. But each kind of concreteness reinforces the other because the goal is the same: to create a programming experience that is more understandable and more directly to the point.

III. Uniformity

Uniformity in a system enables a few concepts to be used to understand everything. For example, uniformity is a startling feature of the physics of the everyday world: all we see about us is constructed out of a few kinds of elementary particles, and everywhere we look the same basic laws of interaction are at play, from the subatomic to the galactic. The Self language strives for a similar kind of relentless uniformity: everything is an object composed of named slots, and all computation is performed by message passing between objects.

The user interface takes a similar approach in its fundamental architecture. We will discuss two kinds of uniformity in the user interface architecture: 1] Morphs all the way down. The user can directly take apart an application down to a very low level. Even the programming environment itself can be modified and deconstructed for use in applications. 2] “Run” and “edit” are unified. There is no system-level distinction between using an application and changing or programming it. Enabling immediate and direct access to pieces of a running application can save time and enhance the sense of direct effect.

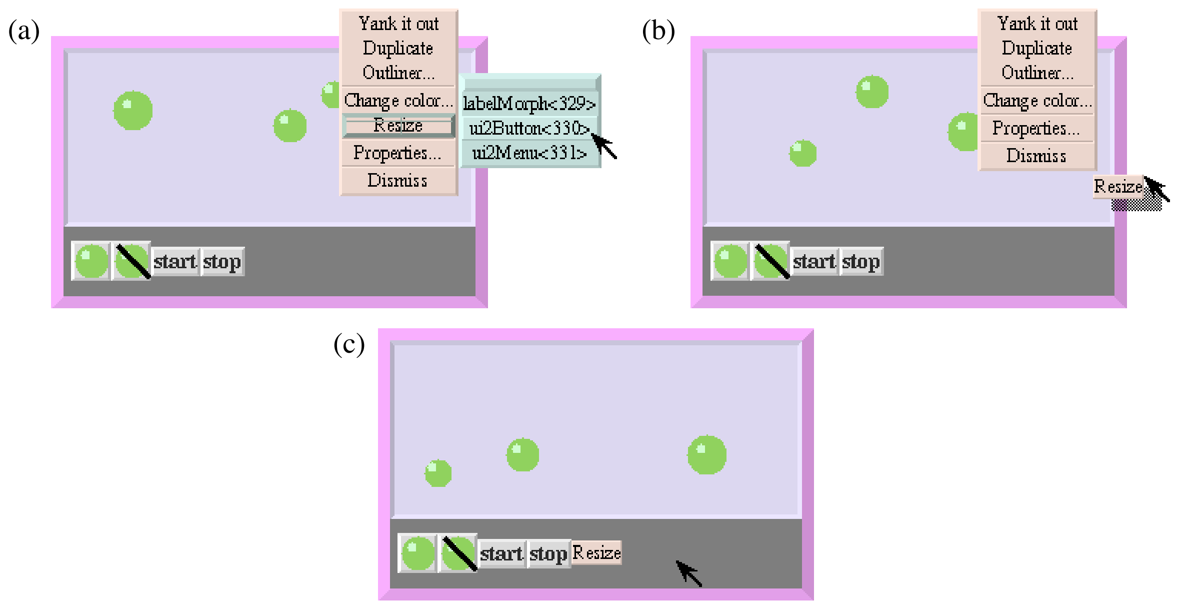


Figure 5. The environment itself is available for reuse. Here the user has created the menu of operations for the gas tank. He has "pinned down" this menu, by pressing a button at the top of the menu (The "pin down" button disappears when pressed.) The user can then take the menu apart into constituent buttons: here he gets the resize button which he then incorporates into the simulation.

1] Morphs all the way down

The user modifying the ideal gas simulation might want to replace the textual label "addAnAtom" appearing on one of the buttons with a picture of an atom. He can extract the label from the submorph structure in a series of interactions depicted in Figure 4.

The user can also "yank out" an atom from the gas tank (as a practical matter he may be want to stop the atomic motion to avoid having to select a moving target), resize the atom to the desired dimension, and drop it onto the button. Dropping alone does not cause embedding: the user selects "embed in morph below" from the atom's meta menu. Again there is a selection ambiguity — embed in *which* morph below? — and the user must select from a menu of morphs at that point in the world. Selecting the button morph from the list embeds an actual atom onto the button.

Environment available for use or modification: An important consequence of uniformity is that the entire environment, being built out of morphs, is available for direct deconstruction or modification. A Self menu, for example, is a kind of column frame with buttons as submorphs. Self menus have a "pin down" bar at the top which enables them to be made persistent. The user pops up the menu for the gas tank, "pins down" this menu, and "yanks out" the resize button. (See Figure 5). The user can then directly embed the resize button into the simulation. Now the simulation user will be able to explore the effects of isothermal volume changes on pressure. In a few gestures, the user has taken an element from the environment for use in the application.

The user can, of course, extend the environment itself in the same very direct way: one might make a new kind of button and embed it in the prototypical systems meta menu, for example. All subsequent meta menu invocations, which copy this menu, will include the new button. As in the physical world, tools are themselves elements of the everyday world, and can be modified using the same mechanisms used to modify other objects.

2] Run and edit unified

There is another kind of uniformity exhibited here: the lack of a run/edit distinction [17]. Most user interface builders have a "build" or "edit mode" that is distinct from the actual application usage mode. (SUIT is a notable exception here [13].) Most interface builders create a description for an application (i.e.: a C program whose execution creates the interface.) There is a deep run/edit switch in the design of such systems, unavoidable in many cases because the underlying system does not support incremental programming changes. But even systems based on Smalltalk or Lisp environments typically do not allow direct, immediate editing of the things on the screen. If you see a menu pop-up as part of an application, we believe there should always be a direct, immediate way to change it. Again, we take inspiration from the underlying laws of the everyday world. Physics does not have distinct run and edit modes, it just keeps right on going. As we have seen through the ideal gas example, there was no fundamental need to stop the animated atoms from bouncing around during construction. This can save time, enhance the sense of direct effect, and it provides immediate feedback.

The run/edit unification is also useful in supporting the multi-user capability of Self-4.0. Any Self window can be shared with any other X window server on the network. Users each get their own cursor and see exactly the same thing, and any object can be manipulated by any user. If one of the users unilaterally decides to start making changes, an environment-wide, or even a per-application run/edit mode could interfere with the other users.

Issues and Limitations

Of course, just as one may want to pull the plug before fixing a toaster, there are certainly times when the user may want to shut down some part or all of an application before modifying it. For example, if one wanted to rewrite the display methods for the atoms in the gas tank, it would be wise to open a new world in which to edit the code, thereby becoming immune from errors that could crash a window's redisplay efforts. Self does its best to make such rude events friendly - a debugger will appear in a separate window anytime a world stops running, but this is an awkwardness worth avoiding.

Using "morphs all the way down" has benefits, but the most fluid reassembly of parts is achieved if the programmer observes a certain design discipline. In particular, morphs need to be built without assuming anything in particular about embedding relationships. For example, a text editor morph might assume there is a scroll bar embedded as the first element in its list of submorphs. Code written under that assumption will not work if the scroll bar is "yanked out." Consequently, morph A, needing to refer to morph B, should maintain a language level slot that holds a reference to morph B. Morphs that are designed within this discipline can keep running even when disassembled.

Summary

Uniformity brings a cognitive economy to operating in the interface: the same mechanisms manipulate and change objects at many levels. Uniformity means that the tools of the environment are like everyday objects, immediately available for use or modification. Finally, uniformity means a lack of distinction between run and edit, which can save time and enhance the sense of immediacy.

IV. Flexibility

In our vision of the ideal system, the programmer could quickly try changes to any aspect of any part of the system. Anything that makes change easier boosts flexibility. In Self-4.0, the morph serves down to a quite low level of graphics detail, and a kind of flexibility emerges: everything the user perceives as a graphical entity is likely to be manipulable as a concrete object. Almost anything can be taken apart and modified, whether it be applications or parts of the environment itself. Uniformity helps flexibility.

Concreteness also aids in flexibility by making it easy to make a change. The user wishing to change an element of the user interface begins by pointing to it and invoking the meta menu.

However, only so much can be done by direct manipulation. Deeper changes require that the user reach into the programming language underlying the user interface.

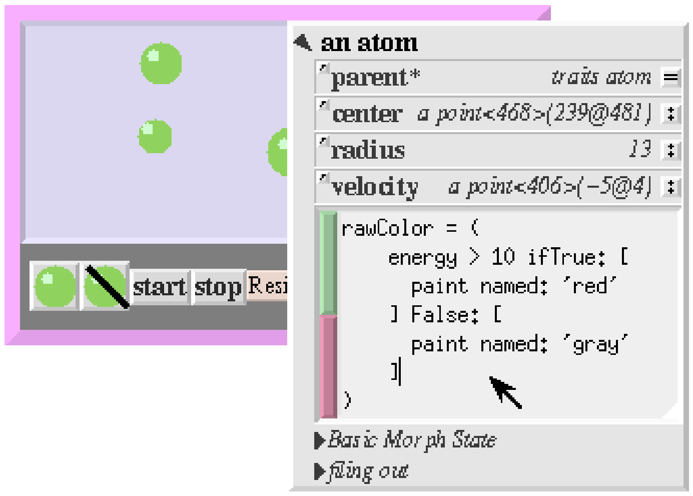
Language level support for flexibility

The uniformity of a language can help flexibility by enabling one way of making changes to apply throughout the system. Uniformity also helps flexibility by allowing the user to combine arbitrary elements from originally distinct domains. For example, a C++ programmer, having built a self-sorting array of objects, is unable to use it immediately for sorting numbers, since numbers are not objects. The flexibility of changing the system to have self-sorting arrays of numbers is impeded because there are two kinds of data. Smalltalk programmers, for whom a number is a first class object, can freely use their sorted collections on numbers or on any other sortable object.

Any language featuring "everything is an object," such as Smalltalk or Self, enjoys this kind of uniformity-generated flexibility. But the Self language provides even more flexibility by two further unifications: 1] use of message-activated slots for both state and behavior, and 2] the lack of a class/non-class distinction. We illustrate both unifications, showing how direct interaction provided by the user interface and programming environment makes such changes easy.

Message-activated slots for both state and behavior.

Self objects use a single construct, the slot, to hold both state and behavior. To be more specific, if a object receives a message and the object holds or inherits a slot matching the message name, then (if the slot contains a non-method object) the object in the slot will be returned as the result of the message send. If the slot contains a method object, the code in the method will be executed, and the result of the execution returned. This uniform usage of slots to hold data and methods makes it very easy to change from using a stored value to a computed value, as no code in the system, even code in slots of the same



```
rawColor = (
    energy > 10 ifTrue: [
      paint named: 'red'
    ] False: [
      paint named: 'gray'
    ]
)
```

Figure 6. The user has selected one atom on which to experiment. The user changes the "rawColor" slot from a computed to a stored value by editing directly in the atom's outliner.

object, can tell if its messages trigger computed or stored results.

To illustrate the environment's support for this kind of change, suppose our user wishes to use color to reveal the energy level of each atom. The idea is to change the atom's color from a stored quantity to a computed one that depends on energy. The user takes a bottom-up approach, planning to test his ideas on an individual atom. By pointing to an atom in the gas tank and selecting the "submorphs" item on the meta menu, the user can summon the atom's outliner.

In the atom's outliner, the user finds a slot (called rawColor) that holds the color of the atom. By selecting "edit" from this slot, the user can change the slot contents from a stored to a computed value. The user puts in some code to return a red color with high energy or gray color with low energy (see Figure 6). (The user must of course understand Self syntax and how the color system works, details of which are off point here.)

There is now one atom running about whose display method uses a computed color, while for the other atoms, which are executing *the very same display method*, color is simply stored. The direct access to an outliner for an atom made it quick and easy to install this change.

No "class/non-class" distinction

Self is a "prototype" based system, and any object can have inheritance children, or can itself inherit from any other object. (For a fuller discussion of the utility of this flexibility, see the original Self paper [18].) In prototype based languages, an object can hold its own behavior, it need not be held on its behalf by some class. The programmer has more places to put behavior, thus more flexibility. In Self, inheriting from an object gives you access to that object's slots. This simple object model enables the Self environment to use a single object representation mechanism, the outliner, to present all the state and behavior available to an object through itself and its parents.

To illustrate how the outliner is used in dealing with inheritance, suppose our user wishes to make *all* the atoms have this kind of energy-based self-coloring behavior. The first step is to make the color calculation more widely available by copying the slot into the shared inheritance "parent" of all atoms. The contents of any slot can be accessed by clicking on the button at the right hand edge of the slot. The user gets an outliner on the shared parent of all atoms from the slot named "parent." From the middle button menu on the slot named "rawColor," the user can select "copy." This attaches a copy of the

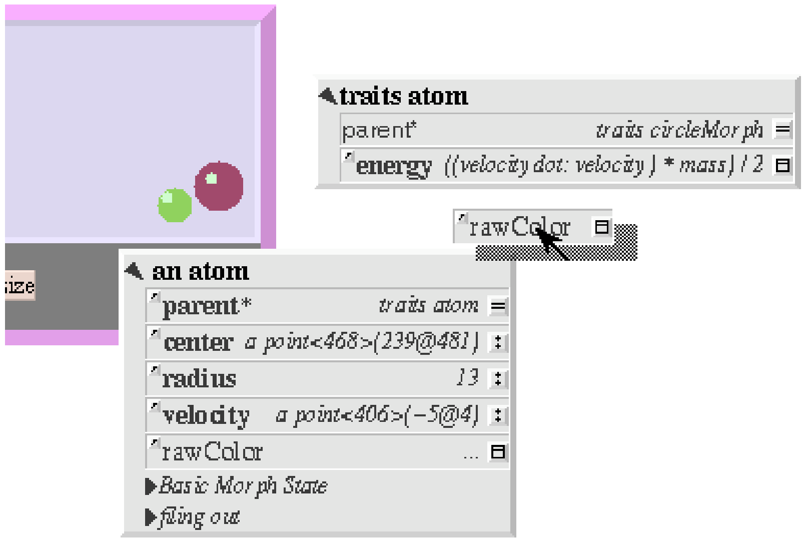

traits atom
parent* traits circleMorph
energy ((velocity dot: velocity) * mass) / 2
rawColor
an atom
parent* traits atom
center a point<468>(239@481)
radius 13
velocity a point<406>(−5@4)
rawColor ...
Basic Morph State
filing out

Figure 7. The user copies the modified rawColor slot as a first step in getting the computed method of coloring more widely shared. Because slots can be moved about readily, restructuring changes are relatively light weight, enhancing the sense of flexibility.

rawColor slot which can be carried and dropped onto the atom parent, who immediately takes it in as one of its own (See Figure 7).

However, this is not enough to make all atoms exhibit this self-coloring behavior, because there is still a rawColor slot in every atom, overriding the new slot in their common parent. The user can remove the rawColor slot from the "prototype" atom. Again, Self is a prototype based system, and consequently the standard way to make a new atom is to copy a broadly available example, or "prototype" atom. In Self, prototypes are stored in the slots of a widely inherited object. The user can get an outliner on this object and browse through the slots to find the one named "atom."

The user can then delete the rawColor slot from the prototype, and all new copies of this atom will be self-coloring. (At this point the gas tank still contains old, original style atoms with their individual "rawColor" slots, so the user may want to empty the gas tank and refill it with the new self-coloring atoms.) The atoms will continue to bounce around throughout the process, and freshly introduced atoms change color in response to being heated or cooled.

To finish the story, user can employ a color changing tool to unify color of the application's morph-submorph collection, move the whole thing to its own window, or save it to a file. (See Figure 8)

Summary

Concreteness plus uniformity are great aids to flexibility, and the independent concreteness and uniformity of the language and the interface give a flexibility to each. But the language (without a distinguished class notion) also helps the environment by enabling one visual element — the outliner — to be used for all programming tasks. The environment, by making slots appear as manipulable concrete entities, makes it easy for the programmer to reorganize things through drag-and-drop, direct-contact based mechanisms. The language works together with the programmer's interface to make a flexible system.

V. The Role of the Implementation

Throughout this discussion, there has been a hidden player whose presence is a necessary part of helping realize the goals of concreteness, uniformity, and flexibility. Previous papers have detailed how the design of Self, so extreme in its insistence on using object and message for everything, renders a conventional implementation unusable [19]. The Self virtual machine uses dynamic compilation based on statistical assessment of actual method usage to generate optimized code while the system runs. This optimization technique inlines many of the calls which would normally be dynamically dispatched. The gas tank simulation for example, takes a few minutes to "warm up" as the system optimizes the code. In the optimized version of the method that calculates how to bounce an atom off the inside of the gas tank wall, the top 44 Self-level stack frames have been inlined into 4 actual frames. Studies of the Self virtual machine have shown that, on the average, there are

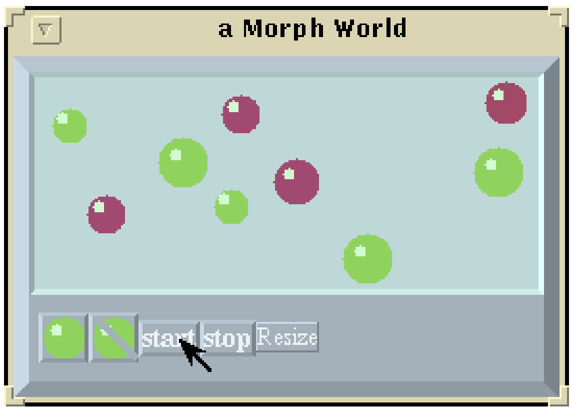


Figure 8. The completed application. The user has invoked a color changing tool that unifies colors across most submorphs, and has invoked the meta menu item "move to own window."

about ten apparent message sends for every one that really happens[9].

Proper support for debugging is an important part of making the programming process fluid and efficient. The Self debugger takes advantage of the virtual machine's technique of "dynamic deoptimization"[8] to present even highly optimized code in Self-level terms. The user can see Self level stack frames, fix any errors in the Self code, and continue.

We believe this implementation lets one work in the style that is naturally called for by the language.

VI. Related work

The Self system draws on previous work in two areas: user interface construction and object-oriented programming languages. It is most closely related to systems that combine both.

User interface builders allow graphical user interfaces to be constructed by direct manipulation. Early work in this area includes Trillium [7], a direct-manipulation editor for photocopier interfaces, Cardelli's dialog editor [3], and the NeXT UI builder [21]. Most UI builders allow the interface to be tested by changing from edit-mode to run-mode; however, even in run-mode, the interface is typically not connected to its underlying application program. To test the UI in the context of the application, the user must recompile or relink and, after testing, must return to the UI builder to make further changes.

The SUIT user interface builder [13] narrows the gap between editing and testing. SUIT uses a special key combination to distinguish editing gestures from normal user interactions. Holding down the SUIT key could be considered a run/edit mode switch, but it is so lightweight and temporary that it does not detract from the sense of liveness and immediacy. However, SUIT does not have an integrated programming environment; the application program itself cannot be changed at run-time.

A number of commercial product such as Hypercard [10], MacroMedia Director [5], and VisualBasic[20] combine a UI builder (perhaps for a specific domain) with a scripting language. User interfaces constructed with these systems can always be edited, and behavior can be added from within the system using the scripting language. However, the scripting languages of these systems have less uniform semantics than Self, and no attempt is made to manifest language elements as concrete, manipulable objects. Furthermore, the scripting languages cannot be used to define user interface widgets nor to extend the system in other ways.

A system called Reno [11], developed independently and concurrently with the Self user interface, has much of the feel we describe here. Display objects are very fine grained, and can be directly accessed at any time. For example, the user can directly embed any graphic into a line of text. Reno is written in Smalltalk, but it is less about an integration of that language with the interface, and more about providing a set of tools for flexible and direct interface manipulation. Reno differs from the Self interface in its interaction details, and in a display metaphor that is based on window objects viewing the contents of container objects.

In the language area, Self is a direct descendant of Smalltalk-80 [6]. Smalltalk achieved some of the goals of Self: the Smalltalk language is extremely uniform in its treatment of objects, its environment allows programs to be changed at any time, and its implementations are fast enough to allow both the application and the user interface framework to be written entirely in Smalltalk. However, the distinction between variable accessing and message sending reduces Smalltalk's language flexibility. Furthermore, Smalltalk's classes are inherently one level abstracted from the objects they describe. The Smalltalk interface, though live, does not provide the kind of concreteness through direct and immediate deconstruction and modification of environment and application that we describe in this paper.

A number of other prototype languages have been developed in recent years including Kevo, Glyphic Script, Omega, and NewtonScript (for overviews, see [16] and [1]). All of these languages have programming environments that allow objects to be inspected and edited graphically. Glyphic Script and NewtonScript also have UI construction facilities. The Newton Toolkit UI builder is similar to other UI builders: the interface is not connected to the application during construction, and a running application cannot be edited.

Glyphic Codeworks takes an interesting approach: constructing a user interface object builds an underlying GlyphicScript object. The fundamental language elements objects and slots are reified and can be manipulated directly. Unfortunately, for lack of a high-performance implementation, most of the Glyphic Codeworks user interface framework, including its user interface components, is implemented in C; these parts of the system cannot be taken apart and modified at run-time.

Garnet [12] and ThingLab [2] are prototype-based user interface construction environments. However, rather than making the underlying programming language more concrete, both systems attempt to hide or at least decrease its importance by using constraints to define user interface behavior. Like Self, Garnet user-interfaces are composed of fine-grained graphical objects like rectangles and circles but, unlike Self, Garnet groups these objects using abstract objects (aggregates) that are not directly visible.

The desktop metaphor [14] made files and directories more approachable by making them into tangible, manipulable objects. The Self system attempts to do the same for an object-oriented programming language. The look and feel of the Self system was heavily influenced by the Alternate Reality Kit (ARK) [15], an earlier system by one of the authors and Seity [4], an earlier user interface for the Self language.

VII. Conclusions

Over the years the Self project has been motivated by a vision of what computing should be. It draws on key properties of the physical world to create a computing system that is more malleable and more readily comprehensible. The key properties, borrowed from the physics of the everyday world, are concreteness, uniformity, and flexibility. Each is present in different ways in the language semantics and the user interface architecture. None would be possible were it not for the dynamically optimizing implementation.

Concreteness is in the language semantics because Self is a prototype-based system: a user works directly on objects, copying them to make new ones. Concreteness is in the user interface architecture in four ways: a physical look and feel, a single Self level representation, the reification of layout constraints, and the use of embedding for composite structure.

Uniformity is in the language semantics because everything is an object, because all computation happens by message passing, and because slots are used to hold both state and behavior. Uniformity is in the user interface architecture because a single kind of display object, used to a very low level, can be directly accessed, and because there is no run/edit distinction. The environment itself is made of morphs, and is therefore available for reuse and modification.

Flexibility in language and interface arises largely from their uniformity and concreteness properties. The user interface helps the language express its flexibility by reifying objects and slots, making them manipulable as concrete entities.

The user interface architecture, the language semantics, and even the implementation all work to create a single wholistic experience motivated by a desire to make programming more like manipulation of tangible objects. More than anything else, one mantra distinguishes this approach from others: concreteness, uniformity, and flexibility. Every aspect of Self—language, implementation, user interface—has been designed to meet what we mean by these three criteria. This coherence of design allows the whole to be more than the sum of its parts.

Acknowledgments

Much credit is due Lars Bak who implemented the outliner, and realized the power of the reified slot. Thanks to Mario Wolczko for the calculation of stack frames in the ideal gas simulation. And thanks to Ole Lehrmann Madsen, Mario Wolczko, and Bay-Wei Chang for helpful comments on the paper.

References

[1] G. Blaschek. **Object-Oriented Programming with Prototypes**, Springer-Verlag, New York, Berlin 1994.

[2] A. Borning and R. Duisberg. *Constraint-Based Tools for Building User Interfaces*, ACM Transactions on Graphics 5(4) pp. 345-374 (October 1981).

[3] L. Cardelli. *Building User Interfaces by Direct Manipulation*, in Proc. ACM Symposium on User Interface Software (UIST '88), pp. 152-166 (October 1988).

[4] B. Chang, D. Ungar, and R. Smith. *Getting Close to Objects*, in M. Burnett, A. Goldberg, and T. Lewis, editors, **Visual Object-Oriented Programming, Concepts and Environments**, pp. 185-198, Manning Publications, Greenwich, CT, 1995.

[5] Director, MacroMedia Corp., San Francisco, CA

[6] A. Goldberg and D. Robson. **Smalltalk-80, the Language and its Implementation**, Addison Wesley, 1983.

[7] D. Henderson. *The Trillium User Interface Design Environment*, Proceedings of CHI '86, pp. 221-227 (April 1986).

[8] U. Hölzle, C. Chambers, and D. Ungar. *Debugging Optimized Code with Dynamic Deoptimization*, in Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, pp. 32-43, San Francisco, CA (June 1992).

[9] U. Hölzle and D. Ungar. *A Third-Generation Self Implementation: Reconciling Responsiveness with Performance*, in Proc. OOPSLA '92, pp. 229-243. Also see U. Hölzle, *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming,* Ph.D. Thesis, Stanford University (August 1994).

[10] HyperCard, Apple Computer Inc., Cupertino, CA.

[11] R. Kerr, M. Markley, M. Sonntag, T. Trower. *Reno: A Component-Based User Interface*, in Proc. CHI '95 Conference Companion, pp 19-20 Denver, (May 1995).

[12] B. Myers, D. Giuse, and B. Vander Zanden. *Declarative Programming in a Prototype-Instance System: Object-Oriented Programming without Writing Methods*, in Proc. OOPSLA '92, pp. 184-200 (October 1992)

[13] R. Pausch, N. Young, R. DeLine. *Simple User Interface Toolkit (SUIT): The Pascal of User Interface Toolkits*, in Proc. Symposium on User Interface Software and Technology (UIST '91), pp. 117-125 (November 1991).

[14] D. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. *Designing the Star user interface*, BYTE 7, 4, pp. 242-282 (April 1982).

[15] R. Smith. *Experiences with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic*, in Proc. CHI + GI Conference, pp 61-67 Toronto, (April 1987).

[16] R. Smith, M. Lenctczner, W. Smith, A. Taivalsaari, and D. Ungar. *Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel)*, in Proc. OOPSLA '94, pp. 102-112 (October 1994). Also see the panel summary of the same title, in Addendum to the Proceedings of OOPSLA '94, pp. 48-53.

[17] R. Smith, D. Ungar, and B. Chang. *The Use Mention Perspective on Programming for the Interface*, In Brad A. Myers, **Languages for Developing User Interfaces,** Jones and Bartlett, Boston, MA, 1992. pp. 79-89.

[18] D. Ungar and R. Smith. *Self: the Power of Simplicity*, in Proc. OOPSLA '87, pp. 227-241 (October 1987).

[19] D. Ungar, R. Smith, C. Chambers, and U. Hölzle. *Object, Message, and Performance: How They Coexist in Self.* Computer, 25(10), pp. 53-64. (October 1992).

[20] VisualBasic, MicroSoft Corp., Redmond, WA

[21] B. Webster. **The NeXT Book**, Addison-Wesley, Reading, MA, 1989.