An Introduction to Morphic:
The Squeak User Interface Framework

John Maloney
DRAFT —— July 20, 2000 —— DRAFT

Introduction

Morphic is a user-interface framework that makes it easy and fun to build lively interactive user interfaces. Morphic handles most of the drudgery of display updating, event dispatching, drag-and-drop, animation, and automatic layout, thus freeing the programmer to focus on design instead of mechanics. Some user interface frameworks require a lot of boiler-plate code to do even simple things. In morphic, a little code goes a long way, and there is hardly any wasted effort.

Morphic facilitates building user interfaces at many levels. At its most basic level, morphic makes it easy to create custom widget. For example, a math teacher could create a morph that continuously shows the x and y components of a vector as the head of the vector is dragged around (Figure 1). Such a widget can be implemented in just 5 methods. The process of creating a new morph is easy, because each aspect of its behavior—its appearance, its response to user inputs, its menu, its drag-and-drop behavior, and so forth—can be added incrementally. Since the class Morph implements reasonable defaults for every aspect of morph behavior, one can start with an empty subclass of Morph and extend its behavior incrementally, testing along the way.

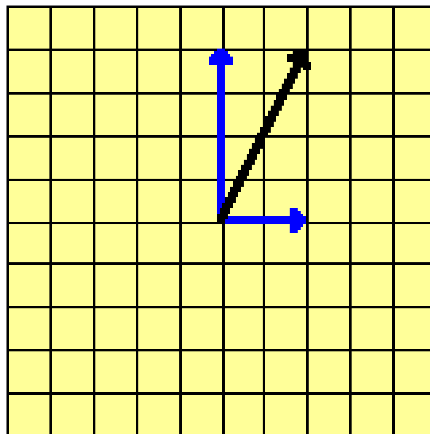

Figure 1: HeadingMorph, a custom widget for showing the x and y components of a vector. This morph can be implemented in just five methods, two for display, two for mouse input, and one for initialization.

At another level, one can work by assembling morphs from the existing library, perhaps using AlignmentMorphs or SystemWindows to arrange them into a single tidy package. Most of the tools of the Squeak programming environment are built by assembling just a few elements—scrolling lists and text editors—into multi-paned windows in a manner similar to the way such tools were created in the old Smalltalk MVC framework. Another example is ScorePlayerMorph, which is just a bunch of sliders, buttons, and strings glued together with alignment morphs. In both these cases, tools are created by assembling instances of pre-existing morph classes. Typically, the component morphs are assembled by the initialization code of the top-level morph (e.g., ScorePlayerMorph). One could imagine constructing this kind of tool graphically by dragging and dropping components from a library and, indeed, morphic was designed with that in mind, although the environmental support needed for this style of tool construction is currently incomplete.

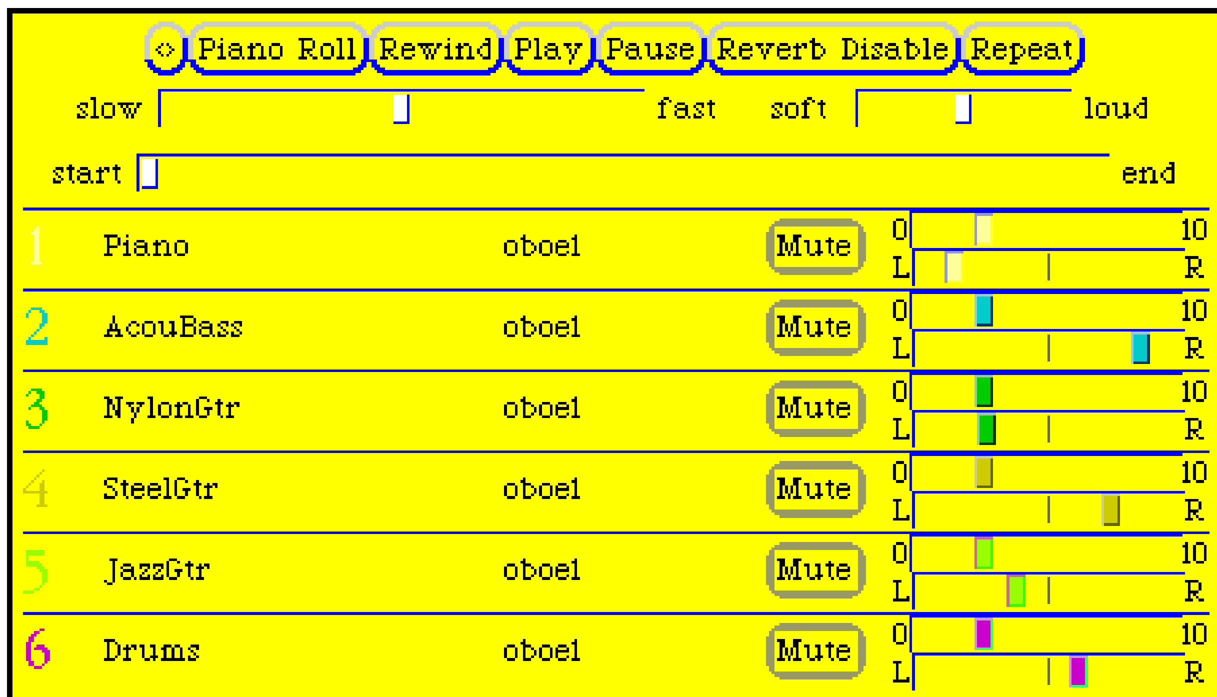

Figure 2: A ScorePlayerMorph, a composite morph assembled from sliders, buttons, strings, and pop-up menu widgets using AlignmentMorphs.

At a more ambitious level, morphic can be used to create viewers and editors for all sorts of information. Examples include SpectrumAnalyzerMorph, which shows a time-varying frequency spectrum of a sound signal in real time (on a reasonably fast computer) and PianoRollMorph, a graphical representation of a musical score that scrolls as the music plays. The relative ease with which such viewers and editors can be created is one of the things that sets morphic apart.

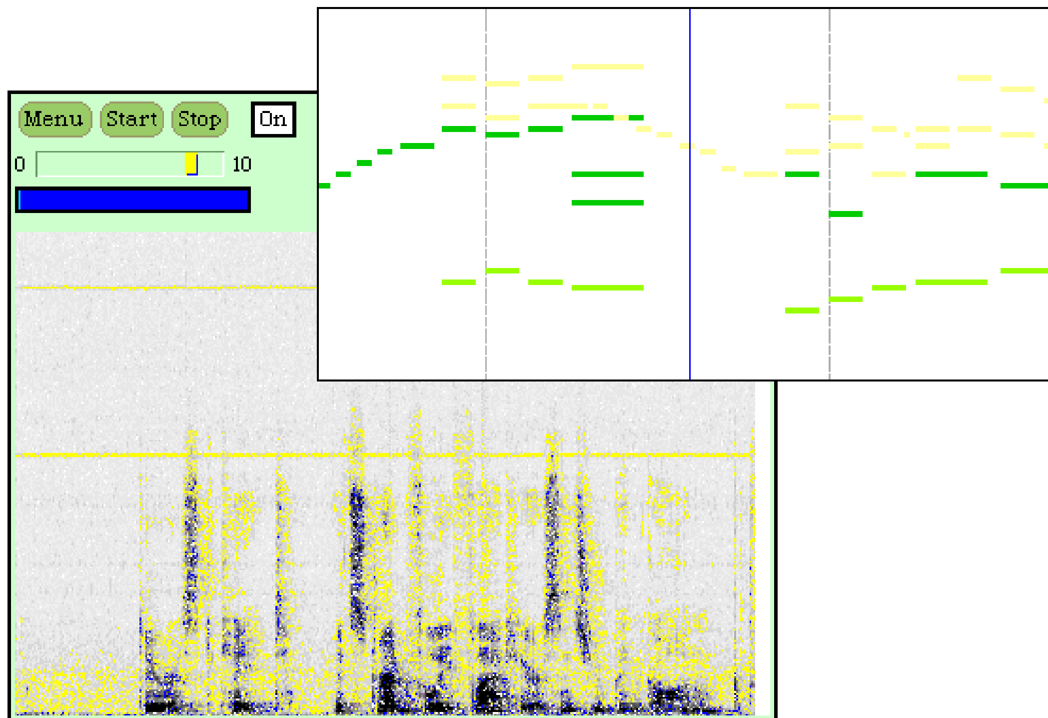

Figure 3: A SpectrumAnalyzerMorph and a PianoRollMorph. These tools go beyond text by presenting dynamic graphical displays of domain-specific data such as a sound spectra or a musical scores. The PianoRollMorph is a simple editor as well as a viewer.

Furthermore, the Squeak's windowing system is itself just a user interface built using morphic. An experienced Morphic programmer could replace all the familiar windows, scroll bars, menus, flaps, and dialogs of Squeak's windowing system with an entirely new look-and-feel. Similarly, one could replace the window system with a custom user interface for a dedicated application such as a CD-ROM game or a palmtop PDA. In short, Morphic was designed to support the full range of user interface construction activities, from assembling pre-existing widgets, to creating new widgets or replacing the window system.

Morphic has two faces. The obvious one is the look and feel of the Squeak development environment itself This chapter, however, focuses on the hidden face of morphic: the one that it presents to the morphic programmer. Once the concepts and principles underlying morphic are understood, it becomes easy to create custom morphs, tools, and entire world's in morphic.

Morphs

The central abstraction of morphic is the graphical object or *morph*. Morphic takes its name from "morph", a Greek word meaning "shape" or "form". The reader may have come across word "morph" in computer graphic contexts,

where it is used as a verb meaning "to change shape or appearance over time." Clearly both uses of the word morph stem from the same Greek root and, although there may be a moment of confusion, the difference between the noun and verb forms of the word quickly becomes clear.

A morph has a visual representation that creates the illusion of a tangible object that can be picked up, moved, dropped on other morphs, resized, rotated, or deleted. Until the recent addition of 3-D to Morphic, morphs were flat objects that could be stacked and layered like a bunch of shapes cut out of paper. When two morphs overlap, one of them covers part of the other morph, creating the illusion that it lies in front of that morph; if a morph has a hole in it, morphs behind it are visible through the hole. This layering of overlapping morphs creates a sense of depth, sometimes called "two-and-a-half-dimensions".. Morphic encourages this illusion be providing a drop shadow when a morph is picked up by the user. This idea, which was used in Randy Smith's Alternate Reality Kit, strengthens the illusion that one is manipulating concrete, tangible objects.

In morphic, the programmer is encouraged to create a new kind of morph incrementally. One might start by defining the morph's appearance, then add interaction, animation, or drag-and-drop behavior as desired. At each step in this process, the morph can be tested and modified. In this section, we will illustrate this incremental development process via a running example. We will create a new kind of morph, then incrementally add methods to define for each aspect of its behavior. While this presentation is not be quite detailed enough to be called a tutorial, the code presented is complete and the interested reader is invited to construct the example morph in their own Squeak system.

Defining a New Morph Class

The first step in creating a custom morph is to define a new empty subclass of Morph:

```
Morph subclass: #TestMorph
    instanceVariableNames: 'angle'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Fun'
```

The angle instance variable will be used later; ignore it for now. Because class Morph defines default behavior for everything a morph must do, one can immediately create an instance of the new morph class for testing by evaluating:

```
TestMorph new openInWorld
```

The new morph appears as simple blue rectangle that can be dragged around. Note that even such a simple morph can be moved, resized, rotated, copied, or

deleted. In short, it is a full-fledged morph merely by virtue of being a subclass of class Morph. That's because class Morph is a concrete class, as opposed to an abstract class that requires that a subclass supply concrete implementation of various methods to make it complete and operational.

Adding Appearance

The programmer can customize the appearance of this new morph by implementing a single method, the "drawOn:" method. In morphic, all drawing is done by sending messages to a drawing engine called a *canvas*. The canvas abstraction hides the details of the underlying graphics system, thus both simplifying programming and providing device independence. While morphic's most heavily used canvas, FormCanvas, uses BitBlt for drawing, it is possible to create subclasses of class Canvas that send graphics commands over a socket for rendering on a remote machine or send Postscript commands to a printer. If color is not available on some device, the canvas can map color to shades of gray or stipple patterns. Once the drawOn: method is written, all these rendering options are available without any additional work.

Many graphic systems use a graphics context object to store such drawing properties as the foreground color and font. This reduces the number of parameters that must be passed in each graphics call. In such a system, the message "fillCircle" might take the fill color, border color, and border width parameters all from the context object. While graphics context style programmer may save a bit on parameter passing overhead, it makes many of the parameters that control a given graphic command are implicit, making it harder to understand the code. Furthermore, since the context state depends on the history of the computation, different execution paths may leave the context in different states, making debugging difficult. In morphic, however, the parameters that control the behavior of a given drawing command—such as the fill color or border width—are passed as explicit parameters to the drawing operations. This makes the code easier to understand and debug.

The best way to learn what graphical operations are available in the canvas protocol is to browse the methods of Canvas. This protocol includes methods that:

- outline or fill rectangles, polygons, curves, and ellipses
- draw lines and single pixels
- draw pixel-based images of any depth
- draw text in any available font and color

You can make the test morph into a colorful oval by drawing it as eight concentric ovals with rainbow colors:

```
drawOn: aCanvas
    | colors |
    colors := color wheel: 6.
```

```
colors withIndexDo: [:c :i |
      aCanvas fillOval: (self bounds insetBy: 4 * i) color: c].
```

The “wheel:” message is sent to the morph’s base color to produce an array of color with the same brightness and saturation, but with six hues spaced evenly around the color wheel. For example, if the morph’s base color is blue, this produces the colors blue, magenta, red, yellow, green, and cyan. The next two lines step through this color array painting concentric ovals, each one inset 4 pixels from the last. To force the morph to redraw itself so you can see the result of this new draw method, just pick it up. When you do this, you’ll also see that its drop shadow is now an oval matching the new shape.

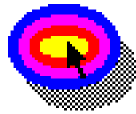

Figure 4: A TestMorph after the drawOn: method has been defined. It has been picked up with the mouse, causing it to cast a shadow.

Note that the result of “self bounds” controls the size of the largest oval. A morph’s bounds is a rectangle that completely encloses that morph. It is used by morphic to support incremental display updates and hit-detection. A morph should never draw outside its bounds. Leaving tracks is probably a symptom of a bug in the morph’s drawOn: method that causes it to draw outside its bounds.

What else can we do to this morph? In morphic, the morph *halo* provides a way to manipulate many aspects of a morph directly. The halo is a constellation of control handles around the morph to be manipulated. Some of these handles can be dragged to change the position or shape of the morph, others perform operations such as copying, deleting, or presenting a menu. The morph halo appears when you hold the ALT key (the CMD or Apple key on a Macintosh) while clicking the mouse on the morph. Place the mouse over a halo handle for a few seconds to get a help balloon that explains what the handle does.

Figure 5: TestMorph with its halo. The cursor has lingered over a handle for a few seconds, causing the help balloon to appear. Moving the cursor away from

the handle will cause the balloon to disappear. Clicking on the background will make the halo itself go away.

Try changing the morph's size using its yellow (lower right) halo handle. Try changing its color by pressing on the red-halo to pop up a menu and selecting the "change color" command. Since the morph's base color is being used as the start of the color wheel, you'll see the entire sequence of colors change. Try duplicating the morph and deleting the duplicate. (If you accidentally delete the original morph, you can make a new instance by evaluating "TestMorph new openInWorld" again.)

Adding Interaction

Morphic represents user actions such as keystrokes and mouse movements using instances of MorphicEvent. A MorphicEvent records both the event type, such as "mouseDown," and a snapshot of the state of the mouse buttons and the keyboard modifier keys (shift, control, and alt or command) at the time that the event occurred. This allows a program to tell, for example, if the shift key was being held down when the mouse button was pressed.

A morph can handle a given kind of event by implementing one of the following messages:

```
mouseDown: evt
mouseMove: evt
mouseUp: evt
keyStroke: evt
```

The event is supplied as an argument so that its state can be examined. The default behavior of the mouseDown: message is to pick up the composite morph containing the morph that gets the event.

To demonstrate interaction in the context of our example, add the following two methods to TestMorph

```
mouseDown: evt
    self position: self position + 10.

handlesMouseDown: evt
    ^ true
```

The first method makes the morph jump 10 pixels down and to the right when it receives a mouseDown event. The second method tells morphic that TestMorph accepts the event. In this case, it accepts all events, but it could accept events selectively based on its internal state or information from the event, such as the shift key state. After adding both methods, click on the morph to try it.

After a few clicks, you'll need to pick up the morph to move it back to it original position. Surprise! You can't just pick up the morph anymore, because it now handles mouse events itself, overriding the default grabbing response. You can use the black halo handle (middle-top) to pick up the morph. (This also works for button morphs or any morph that has its own mouseDown behavior.) Another way to handle this problem is to reject mouseDown events if the shift key is down thus allowing the morph to be grabbed:

```
handlesMouseDown: evt
	^ evt shiftPressed not
```

Now you can just hold down the shift key when you want to pick up the morph.

In addition the basic mouse events, morphs can handle events when the cursor enters and leaves a morph, either with or without the mouse down, or it can elect to get click and double-click events. To learn more about these kinds of events, browse the "event handling" category of Morph.

Adding Drag and Drop

A morph can perform some action when another morph is dropped onto it, and it can decide which dropped morphs it wishes to accept. To accept dropped morphs, a morph must answer true to the message:

```
wantsDroppedMorph:event:
```

The morph being dropped is supplied as an argument to allow the receiving morph to decide if it wishes to accept it. For example, a printer icon morph in a desktop publishing application might accept only morphs representing printable documents. The event is also supplied so that the modifier keys at the time of the drop are available. If the receiving morph agrees to accept the dropped morph, it is then sent the message:

```
acceptDroppingMorph:event:
```

to actually perform the drop action. For example, a printer morph might queue a print request when a document morph is dropped onto it.

After the recipient of the dropped morph has processed the drop action, the dropped morph itself might need to perform some action. The dropped morph is informed that it has been dropped by sending it the message:

```
justDroppedInto:event:
```

The morph that accepted the drop is provided as an argument and the triggering event (typically a mouse up event) is provided so that the modifier keys at the time of the drop are available. Most of the time, the default behavior is appropriate, so the programmer need not implement this method.

To demonstrate drag and drop in the context of the example, add the following two methods to TestMorph

```
acceptDroppingMorph: aMorph event: evt
    self color: aMorph color.

wantsDroppedMorph: aMorph event: evt
    ^ true
```

To test it, create some RectangleMorphs (using the "New Morph" item of the background menu), give them various colors (using the "change color..." command in the "fill style" submenu of the red halo menu), and drop them on your TestMorph.

Adding Liveness

Animation makes an interactive application come alive and can convey valuable information. However, graphical animation—objects moving and changing their appearance over time—is just one aspect of a more general user interface goal we call "liveness". Other examples of liveness include user interface objects that update themselves dynamically—such as inspectors, clocks, or stock quote displays—and controls that act continuously on their targets, like the morphic color chooser.

Animation in a user interface becomes annoying if the user is locked out while the animation runs. In morphic, liveness and user actions are concurrent: any number of morphs can be animated and alive, even while the user drags a scroll bar or responds to a typein prompt. In his August, 1981 Byte article, Larry Tesler argued that the system should not lock the user in a mode. Morphic goes one step further: it also keeps the user from locking the system into a mode.

Let's animate our TestMorph example by making it go in a circle when clicked. Add the following three methods:

```
mouseDown: evt
    angle := 0.
    self startStepping.

step
    angle := angle + 5.
    angle >= 360 ifTrue: [^ self stopStepping].
    self position: self position + (Point r: 8 degrees: angle).
```

Now you see why we needed the 'angle' instance variable: to store the current direction for this animation. The 'mouseDown:' method initializes the angle to zero degrees and asks morphic to start sending step messages to the morph. The liveness of a morph is defined by its step method. This step method advances the angle by five degrees. It then checks to see if the circle is

complete and, if so, it tells morphic to stop sending step messages. If the circle isn't complete, the morph updates its position by moving 8 pixels in the direction of the current angle.

Click on the morph to try this. You'll notice that it moves, but very slowly. In fact, 'step' is being sent to it at the default rate of once per second. To make the morph go faster, add the method:

```
stepTime
    ^ 20
```

On a fast enough machine, the morph will be sent step every 20 milliseconds, or 50 times a second, and it will really zip. Here's one more thing to try: make several copies of the morph (using the green-halo handle) and quickly click on all of them; you will see that multiple animations can proceed concurrently and that you can still interact with morphs while animations run.

There are several things to keep in mind about step methods. First, since they may run often, they should be as efficient as possible. It is also a good to use the appropriate step time. A clock morph that only displays hours and minutes need not be stepped more often than once a minute (a step time of 60,000 milliseconds). Finally, the stepTime method only defines the minimum desired time between steps; there is no guarantee that the time between steps won't be longer. Applications that wish pace themselves against real-world time can do interpolation based on Squeak's millisecond clock. The Alice 3-D system does exactly this in order to support time-based animations such as "make a complete circle in three seconds."

Example: PicoPaint

As a final example, this section shows how the core of the simple sketch editor shown in Figure 6 can be implemented in only six methods totaling about 30 line of code.

Figure 6: Drawing a picture with PicoPaintMorph, a simple sketch editor.

The first step is, as usual, to create an empty subclass of class Morph:

```
Morph subclass: #PicoPaintMorph
    instanceVariableNames: 'form brush lastMouse '
```

```
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Morphic-Fun'
```

The instance variable 'form' will hold the sketch being edited, an instance of class Form, 'brush' will be a Pen on that Form, and 'lastMouse' will be used during pen strokes. Implementing the 'extent:' method allows the sketch to be resized, so the user can make it whatever size they like:

```
    extent: aPoint
        | newForm |
        super extent: aPoint.
        newForm := Form extent: self extent depth: 16.
        newForm fillColor: Color veryLightGray.
        form ifNotNil: [form displayOn: newForm].
        form := newForm.
```

This method is invoked whenever the morph is resized, such as when the user drags the yellow halo handle. Note that a Form of the new size must be created and that the contents of the old sketch copied into it. To make sure that we start off with a sketch of some default size, we override the 'initialize' method:

```
    initialize
        super initialize.
        self extent: 200@150.
```

Note that both 'extent:' and 'initialize' start by invoking the default versions of these methods inherited from class Morph. These inherited methods handle all the morphic bookkeeping details so the programmer of the subclass doesn't have to worry about them.

Now that the 'form' instance variable is initialized and maintained across size changes, adding draw method is trivial:

```
    drawOn: aCanvas
        aCanvas image: form at: bounds origin.
```

At this point, one could create an instance of PicoPaintMorph and it would appear as a light gray rectangle that can be resized. To make it into a sketch editor, we just need to add user input behavior to draw a stroke when the mouse is pressed on the morph. This requires three methods:

```
    handlesMouseDown: evt
        ^ true

    mouseDown: evt
        brush := Pen newOnForm: form.
        brush roundNib: 3.
        brush color: Color red.
```

```
        lastMouse := evt cursorPoint - self position.
        brush drawFrom: lastMouse to: lastMouse.
        self changed.

    mouseMove: evt
        | p |
        p := evt cursorPoint - bounds origin.
        p = lastMouse ifTrue: [^ self].
        brush drawFrom: lastMouse to: p.
        lastMouse := p.
        self changed.
```

The 'mouseDown:' method creates a Pen on the sketch Form and draws a single point at the place where the mouse went down. Note that mouse event positions are in world coordinates which must be converted into points relative to the origin of the sketch Form before using them to position the pen. The mouseMove: method uses the 'lastMouse' instance variable to decide what to do. If the mouse hasn't moved, it does nothing. If the mouse has moved, it draws a stroke from the previous mouse position to the new mouse position and updates the 'lastMouse' instance variable.

Note that both the 'mouseDown:' and 'mouseMove:' methods end with 'self changed'. This tells morphic that the morph's appearance has changed so it must be redrawn. But if you make the sketch very large and draw a circle quickly, you will notice that the circle drawn by the pen is not smooth, but a rather coarse approximation made of straight line segments. The problem is more pronounced on slower computers. Yet if the sketch is made small, the problem is less severe. What is going on?

This is a performance problem stemming from the fact that morphic's incremental screen updating is redrawing the entire area of the display covered by the sketch. As the sketch gets larger, the display updating takes more time, and thus the morph can't processes as many mouse events per second. Fortunately, it is easy to improve matters by noticing that only a portion of the sketch must be updated with each mouse event: namely, the rectangle spanning the last mouse position (if any) and the current one. If mouse only moves a few pixels between events, the portion of the display to be updated is small. By reporting only this small area, rather than the area of the entire sketch, we can make drawing performance independent of the size of the sketch:

```
    mouseDown: evt
        brush := Pen newOnForm: form.
        brush roundNib: 3.
        brush color: Color red.
        lastMouse := evt cursorPoint - self position.
        brush drawFrom: lastMouse to: lastMouse.
        self invalidRect:
```

```
            ((lastMouse - brush sourceForm extent corner: lastMouse +
brush sourceForm extent)
                translateBy: self position).

mouseMove: evt
    | p |
    p := evt cursorPoint - bounds origin.
    p = lastMouse ifTrue: [^ self].
    brush drawFrom: lastMouse to: p.
    self invalidRect: ((
        ((lastMouse min: p) - brush sourceForm extent) corner:
        ((lastMouse max: p) + brush sourceForm extent))
            translateBy: bounds origin).
    lastMouse := p.
```

The 'invalidRect;' method takes a damage rectangle in screen coordinates. This rectangle is expanded on all sides by the size of the pen nib. (Actually, a square nib extends down and to the right of its position, while a circular nib is centered at its position. For the sake of simplicity, this code invalidates a slightly larger rectangle than strictly necessary.)

It's easy to extend this sketch editor with menu commands to change the pen size and color, clear the sketch (actually, this can be done already by using the yellow halo handle to shrink and re-expand the sketch editor), fill outlines with a color, read and write sketch files, and so on.

Composite Morphs

Like most user interface tool kits and graphic editors, morphic has a way to create composite graphical structures from simpler pieces. Morphic does this using embedding: any morph can embed other morphs to create a composite morph. Figure 7 shows the result of embedding button, string, and star morphs in a rectangle morph.

Figure 7: A composite morph created by embedding various morphs in a rectangle morph. The composite morph is being moved. The embedded morphs stick out past the edge of the rectangle, which is reflected by the drop shadow.

A composite morph structure behaves like a single object—if you pick it up and move it, you pick up and move the entire composite morph. If you copy or delete it, the entire composite morph is copied or deleted.

The glue that binds sub-assemblies together in many graphics editors is intangible , merely the lingering after effect of applying the "group" command to a set of objects. In contrast, the binding agents in a composite morph are concrete morphs. If a composite morph is disassembled, each of its component morphs is a concrete morph that can be seen and manipulated. This allows composite morphs to be assembled and disassembled almost like physical objects.

Morphic could have been designed to have two kinds of morph: atomic morphs and grouping morphs. But in a sense, this would be like the "grouping command" approach. What would be the visual manifestation of a group morph? If it were visible, say as an outline around its submorphs, it would be a visual distraction. This suggests that group morphs should be invisible. Yet if all the morphs were removed from a group morph, it would need some sort of visual manifestation so it could be seen and manipulated. Morphic neatly avoids this issue by having *every* morph be a group morph. For example, to create a lollipop, one can just embed a circle morph on the end of a thin rectangle morph. Reversing that operation makes the two morphs independent again. It feels concrete, simple, and obvious.

At this point, some terminology may be useful. The morphs embedded in a composite morph are called its *submorphs*. A submorph refers to the morph in which it is embedded as its *owner*. The terms submorph and owner describe relationships between morphs, not kinds of morphs. Any morph can contain submorphs, be a submorph, or both at once. The base of a composite morph structure it called its *root*.

Of course, those with computer science backgrounds will immediately realize that the structure of a composite morph is a tree. Each morph in this tree knows both its owner morph and all of its submorphs. While morphic could have been designed so that morphs did not know their owners, one of morphic's design goals was that a morph should be able to find out about its environment. This makes it simpler for objects in a simulation to find out about—and respond to—their environment. For example, in a billiards simulation, the morph representing the cue stick might go up its owner chain to find the billiards table morph, and from there find all the billiard balls on the table.

The morphs on the screen are actually just submorphs of a morph called the "world" (actually, an instance of PasteUpMorph). The object representing the user's cursor is a morph called the hand (HandMorph). A morph is picked up by removing it from the world and adding it to the hand. Dropping the reverses this process. When a morph is deleted, it is removed from its owner and its owner is set to nil. The message "root" can be sent to a morph to

discover the root of the composite morph that contains it. The owner chain is traversed until a morph whose owner is a world, hand, or nil is encountered; that morph is the root.

How does one construct a composite morph? In the morphic programming environment, it is easy. One just places one morph over another and invokes the “embed” command from the halo. This makes the front morph become a submorph of the morph immediately behind it. When writing code, the “addMorph:” operation is used. In either case, adding a submorph updates both the owner slot of the submorph and the submorphs lists of its old and new owner. For example, adding morph B to morph A adds B to A’s submorph list, removes B from its old owner’s submorph list, and sets B’s owner to A. The relative positioning of the two morphs is not changed by this operation unless the new owner does some kind of automatic layout.

Automatic Layout

Automatic layout relieves the programmer from much of the burden of laying out the components of a large composite morph such as the ScorePlayerMorph shown in Figure 2. By allowing morphic to handle the details of placing and resizing, the programmer can focus on the topology of the layout—the ordering and nesting of submorphs in their rows and columns—without worrying about their exact positions and sizes. Automatic layout allows composite morphs to adapt gracefully to size changes, including font size changes. Without some form of automatic layout, changing the label font of a button might require the programmer to manually change the size of the button and the placement of all the submorphs around it.

Layout Morphs

Most morphs leave their submorphs alone; the submorphs just stay where they are put. However, certain morphs, called *layout morphs*, actively control the placement and size of their submorphs. The most common such layout morph, AlignmentMorph, employs a simple layout strategy: linear, non-overlapping packing of its submorphs along a single dimension. A given AlignmentMorph can be set to pack either from left-to-right or from top-to-bottom, making it behave like either a row or column. Although this layout strategy does not handle every conceivable layout problem (it’s not great for spreadsheet-like tableaus), it does cover a surprisingly wide range of common layout problems. A morphic programmer could also create layout morphs using other packing algorithms if necessary.

Linear packing is best explained procedurally. The task of a horizontal AlignmentMorph is to arrange its submorphs in a row such that the left edge of each morph just touches the right edge of the next morph. Submorphs are processed in order: the first submorph is placed at the left end of the row, then the next submorph is placed just to the right of the first, and so on. Notice that packing is done only in one primary dimension—the horizontal dimension

in this case. Placement along the secondary dimension is controlled by the justification field of the AlignmentMorph. In our example, this field determines whether submorphs are placed at the top, bottom, or center of the row.

Space Filling and Shrink Wrapping

For simplicity, the packing strategy was described as if the submorphs being packed were all rigid. In order to support "stretchy" layouts, an AlignmentMorph can be designated as space-filling. When there is extra space during packing, any space-filling AlignmentMorph submorphs expand to fill this space. When there is no extra space, a space-filling morph shrinks to its minimum size. When there are several space-filling morphs in a single row or column, any extra space is divided evenly among them.

Space-filling AlignmentMorph can be added to control the placement of other submorphs within a row or column. For example, suppose one wanted a row with three buttons, one at the left end, one at the right end, and one in the middle. This can be accomplished by inserting space-filling AlignmentMorphs between the buttons as follows:

<button one><space-filler><button two><space-filler><button three>

The result is shown in figure 8. When the row is stretched, the extra space is divided evenly between the two space-filling morphs, so that button one stays at the left end, button two stays centered, and button three gets pushed to the right end.

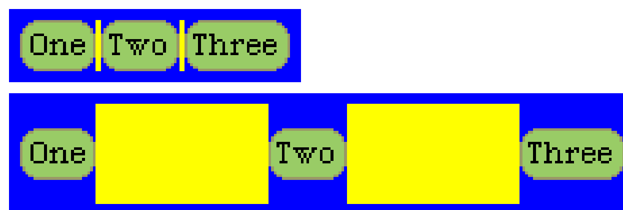

Figure 8: Using space-filling AlignmentMorphs (light gray) to distribute button morphs evenly within a row. The inset attribute of the row was set to leave a little extra space around its edges. The row is shown at its minimum size and at a larger size. For clarity, the space filling morphs have been made a contrasting color; normally, they would be the same color as the row, making them effectively invisible.

It is sometimes desirable for the size of an AlignmentMorph to depend on the size of its submorphs. For example, a labeled box should depend on the size of its label so that it automatically resizes itself when its label changes. An alignment morph designated as shrink-wrap contracts or grows to the smallest size that accommodates the space requirements of its submorphs.

Layout Attributes

AlignmentMorph has a number of attributes that control how layout is done. The **orientation** attribute, which determines whether the AlignmentMorph lays out its submorphs in a row or column, can be set to either **horizontal** or **vertical**.

The **centering** attribute controls centering in the non-layout dimension. It can be set to:

center	Submorphs are centered within the row or column.
topLeft	Submorphs are aligned along the top of a row or the left. edge of a column.
bottomRight	Submorphs are aligned along the bottom of a row or the right edge of a column.

AlignmentMorph has separate resize attributes for the horizontal (**hResizing**) and vertical (**vResizing**) dimension; the two dimensions are completely independent in their resizing behavior. These can be set to:

rigid	This morph is never resized.
spaceFill	When this morph is the submorph of another AlignmentMorph, this morph expands or shrinks depending on the space available. Extra space is distributed evenly among all space-filling morphs in a row or column.
shrinkWrap	This morph is shrunk to just fit around its submorphs, or to its minimum size, whichever is smaller. Any enclosed space-filling morphs are shrunk as needed.

How Morphic Works

This section gives an overview of how morphic works in just enough detail to help the morphic programmer get the most out of the system.

The UI Loop

At the heart of every interactive user interface framework lies the modern equivalent of the read-evaluate-print loop of the earliest interactive computer systems. However, in this modern version, “read” processes events instead of characters and “print” performs drawing operations to update a graphical display instead of outputting text. Morphic’s version of this loop adds two additional steps to provide hooks for liveness and automatic layout:

```
do forever:
    process inputs
```

send “step” to all active morphs
update morph layouts
update the display

Sometimes, none of these steps will have anything to do; no events to process, no morph that needs to be stepped, no layout updates, and no display updates. In such cases, morphic sleeps for a few milliseconds so that it doesn’t hog the CPU when its idle.

Input Processing

Input processing is a matter of any dispatching incoming events to the appropriate morphs. Keystroke events are sent to the current keyboard focus morph, which is typically established by a mouse click. If no keyboard focus has been established, the keystroke event is discarded. There is at most one keyboard focus morph at any time.

Mouse down events are dispatched by location; the front-most morph at the event location gets to handle the event. Events do not pass through morphs; you can’t accidentally press a button that’s hidden behind some other morph. Morphic needs to know which morphs are interested in getting mouse events. It does this by sending each candidate morph the “handlesMouseDown:” message. The event is supplied so that a morph can decide if it wants to handle the event based on which mouse button was pressed and which modifier keys were held when the event occurred. If no morph can be found to handle the event, the default behavior is to pick up the morph.

Within a composite morph, the front-most submorph is given the first chance to handle the event, consistent with the fact that submorphs appear in front of their owner. If that submorph does not want to handle the event, its owner is given a chance. If its owner doesn’t want it, then the owner’s owner gets a chance, and so on, up the owner chain. This policy allows a mouse sensitive morph, such as a button, to be decorated with a label or graphic and still get mouse clicks. In the first attempt at event dispatching, mouse clicks on button labels were not passed on to the owning button, so clicks that hit the label were blocked, and it not so easy to click on a button without hitting its label!

What about mouse move and mouse up events? Consider what happens when the user drags the handle of a scroll bar. When the mouse goes down on the scroll bar, the scroll bar starts tracking the mouse as it is dragged. It continues to track the mouse even if the cursor moves outside of the scroll bar, even if the cursor is dragged over a button or some other scroll bar. That is because morphic considers the entire sequence of mouse down, repeated mouse moves, and mouse up to be a single “transaction”. Whichever morph accepts the mouse down event is considered the “mouse focus” until the mouse goes up again. This mouse focus morph is guaranteed to get the entire mouse drag transaction: a mouse down event, at least one mouse move event, and a mouse up event. Thus, a morph can perform some initialization on mouse down and

cleanup on mouse up, and be assured that the initialization and cleanup will always get done.

Stepping

Stepping is handled by keeping a list of morphs that need to be stepped, along with their desired next step time. Every cycle, the “step” message is sent to any morphs that are due for stepping and their next step time is updated. Deleted morphs are pruned from the step list, both to avoid stepping morphs that are no longer on the screen, and to allow those morphs to be garbage collected.

Layout Updating

Morphic maintains morph layout incrementally. When a morph is changed in a way that could influence layout (e.g., when a new submorph is added to it), the message layoutChanged is sent to it. This triggers a chain of activity. First, the layout of the changed morph is updated. This may change the amount of space given to some of its submorphs, causing their layouts to be updated. Then, if the space requirements of the changed morph have changed (e.g., if it needs more space to accommodate the newly added submorph), the layout of its owner is updated, and possibly its owner’s owner, and so on. In some cases, the layout of every submorph in a deeply-nested composite morph may need to be updated. Fortunately, there are many cases where layout updates can be localized, thus saving a great deal of unnecessary work.

As with changed messages, morph clients usually need not send layoutChanged explicitly since the most common operations that affect the layout of a morph—such as adding and removing submorphs or changing the morph’s size—do this already. The alert reader might worry that updating the layout after adding a morph could make building row or column with lots of submorphs inefficient. In fact, if the cost of updating the layout is proportional to the number of morphs already in the alignment morph, then adding N morphs one at a time would have a cost proportional to N^2! To avoid this very problem, morphic defers all layout updates until the next display cycle. After all, you can’t see the layout change until the screen is next repainted anyhow. Thus, a program can perform any number of layout-changing operations on a given morph between display cycles and morphic will do only update that morph’s layout once.

Display Updating

Morphic uses a double-buffered, incremental algorithm to keep the screen updated. This algorithm is efficient (it tries to do as little work as possible to update the screen after a change) and high-quality (the user does not see the screen being repainted). It is also mostly automatic; many applications can be built without the programmer ever being aware of how the display is maintained. The description here is mostly for the benefit of those curious about how the system works.

Morphic keeps a list, called the “damage list” of those portions of the screen that must be redrawn.. Every morph has a bounds rectangle that encloses its entire visible representation. When a morph changes some aspect appearance (for example, its color), it sends itself the message ‘changed’, which adds its bounding rectangle to the damage list. The display update phase of the morphic UI loop is responsible for bringing the screen up to date. For each rectangle in the damage list, it redraws (in back-to-front order) all the morphs intersecting the damage rectangle. This redrawing is done in an off-screen buffer which is then copied to the screen. Since individual morphs are drawn off screen, the user never sees the intermediate stages of the drawing process, and the final copy to the screen is quite fast. The result is smooth animation of objects that seem solid regardless of the sequence of individual drawing operations. When all the damage rectangles have been processed, morphic clears the damage list to prepare for the next cycle.

Design Principles Behind Morphic

The design principles behind a system—why things are done one way and not some other way—are often not manifest in the system itself. Yet understanding the design philosophy behind a system like morphic can help programmers extend the system in ways that are harmonious with the original design. This section articulates three important design principles underlying morphic: concreteness, liveness, and uniformity.

Concreteness

We live in a world of physical objects that we constantly manipulate. We take a book from a shelf, we shuffle through stacks of papers, we pack a bag. These things seem easy because we’ve internalized the laws of the physical world: objects are persistent, they can be moved around, and if one is careful about how one stacks things, they generally stay where they are put. Morphic strives to create an illusion of concrete objects within the computer that has some of the properties of objects the physical world. We call this principle *concreteness*. Concreteness helps the morphic user understand what happens on the screen by analogy with the physical world. The user quickly realizes that everything on the screen is a morph that can be touched and manipulated. Pixels are not dribbled onto the screen by some long gone process or procedure; rather, the agent that wrote those pixels is always a morph that can be investigated and manipulated. Compound morphs can be disassembled and individual morphs can be inspected, browsed, and changed. Since all these actions begin by pointing directly at the morph in question, we sometimes say that *directness* is another of morphic design principles. Concreteness and directness create a strong sense of confidence and empowerment; the user quickly gains the ability to reason about morphs the way they do about physical objects.

Morphic achieves concreteness and directness in several ways. First, the display is updated using double-buffering, so the user never sees morphs in

the process of being redrawn. Unlike some user interfaces, in which an object being moved is shown only as an outline, morphic always show the full object. In addition, when an object is picked up, it throws a translucent drop shadow the exact shape as itself. Taken together, these display techniques create the sense that morphs are flat physical objects, like shapes cut out of paper, lying on a horizontal surface until picked up by the user. Like pieces of paper, morphs can overlap and hide parts of each other, and they can have holes that allow morphs behind them to show through.

Second, every pixel on the morphic display is put there by some morph. If a morph is moved or deleted, the display is updated immediately. (Of course, in Smalltalk it is always possible to draw directly to the Display, but the concreteness of morphs is so nice that there is high incentive to write code that plays by the morphic rules.) Since a morph draws only within its bounds, it is always possible to locate the morph responsible for something seen on the display by pointing at it.

Halos allow many of aspects of a morph—its size, position, rotation, and composite morph structure—to be manipulated directly by dragging handles on the morph it self. This is sometimes called "action-by-contact." In contrast, some user interfaces require the user to manipulates objects through menus or dialog boxes that are physically remote from the object being manipulated, which might be called "action-at-a-distance." Action-by-contact reinforces directness and concreteness; in the physical world, we usually manipulate objects by contact. Action-at-a-distance is possible in the real world—you can blow out a candle without touching it, for example—but such cases are uncommon and feel like magic.

Finally, as discussed earlier, concrete morphs combine directly to produce composite morphs. If you remove all the submorphs from a composite morph, the parent morph is still there as a concrete morph. No invisible "composite morph" or "glue" objects hold submorphs together; all the pieces are concrete, and the composite morph can be re-assembled again by direct manipulation. The same is true for automatic layout—layout is done by morphs that have a tangible existence independent of the morphs they contain. Thus, there is a place one can go to understand and change the layout behavior. We say that morphic *reifies* composite structure and automatic layout behavior.

Liveness

Morphic is inspired by another property of the physical world: *liveness*. Many objects in the physical world are active: clocks tick, traffic lights change, phones ring. Similarly, in morphic any morph can have a life of its own: object inspectors update, piano rolls scroll, blobs crawl around. Just as in the real world, morphs continue to run while the user does other things. In stark contrast to user interfaces that wait passively for the next user action, morphic becomes an equal partner in what happens on the screen. Instead of

manipulating dead objects, the user interacts with live ones. Liveness makes morphic fun.

Liveness allows the use of animation in the interface. For example, if one drops an object on something that doesn't accept it, it can animate smoothly back to its original position. Liveness also supports a useful technique called *observing*, in which some morph (e.g., an UpdatingStringMorph) presents a live display of some data. Unlike notification-based schemes like MVC, in which the view watches a model that's been carefully instrumented to send change reports, observing allows one to watch things that were not designed to be watched. For example, while debugging a memory-hungry multimedia application, one might wish to watch total bytes of memory used by Form objects. While this is not a quantity already maintained by the system, it can be computed and observed. Even things outside of the Squeak system can be observed, such as the number mail messages on the mail server.

Observing is a polling technique—the observer periodically compares its current observation with the previous observation and performs some action when they differ. This does not necessarily mean it is inefficient. First, the observer only updates the display when the observed value changes, so the is no display update cost for when the value doesn't change. Second, the polling frequency of the observer can be adjusted. Even if it takes a tenth of a second to compute the bytes used by all Form objects, if this computation is done once per minute, it will consume well under one percent of the CPU cycles. Of course, a low polling rate creates a time lag before the display reflects a change, but this loose coupling also allows rapidly changing data to be observed (sampled, actually) without reducing the speed of the computation to the screen update rate.

The primary mechanism used to achieve liveness the *stepping* mechanism. As we saw, any morph can implement the "step" message and can define its desired step frequency. This gives morphs a heartbeat that they can use for animation, observing, or other autonomous behavior. It is surprising that such a simple mechanism is so powerful. Actually, the stepping mechanism alone is not enough. Liveness is also enabled by morphic's incremental display management, which allows multiple morphs to be stepping at once without worrying about how to sequence their screen updates. Morphic's support for drag-n-drop and mouse-over behavior further add to the sense of system liveness.

Morphic avoids the global run/edit switch found in many other systems. Just as you don't have to (and can't!) turn off the laws of physics before manipulating an object in the real world, you needn't suspend stepping before manipulating a morph or even editing its code. Things just keep running. When you pop up a menu or halo on an animating morph, it goes right on animating. When you change the color of a morph using the color palette, its color updates continuously. If you're quick enough, you can click or drop

something on an animating morph as it moves across the screen. All these thing support the principle of liveness.

Uniformity

Yet another inspiring property of the physical world is its uniformity. No matter where you go and what you do, objects obey the same old physical laws. We use this uniformity every day to predict how will behave in new situations. If you drop an object, it falls; you needn't test every object you come across to know that it obeys the law of gravity.

Morphic strives to create a similar uniformity for objects on the screen, a kind of "physics" of morph interactions. This helps users reason about the system and helps them put morphs together in ways not anticipated by the designers. For example, since menus in morphic are just composite morphs, one could extract a few handy commands from a menu and embed them in some other morph to make a custom control panel. Uniformity is achieved in morphic by striving to avoid special cases. Everything on the screen is a morph, all morphs inherit from class Morph, any morph can have submorphs or be a submorph, and composite morphs behave like atomic morphs: in these and other design choices, morphic seeks ways to merge different things under a single general model and avoids making distinctions that would undermine uniformity.

The Past and Future of Morphic

The first version of morphic was developed by John Maloney and Randy Smith at Sun Microsystems Laboratories as the user interface construction environment for the Self 4.0 system. Self is a prototype-based language, similar to Smalltalk but without classes or assignment. (Self uses message sends to access variables). Randy's previous work with the Alternate Reality Kit, and his passion for concreteness and uniformity, heavily influenced the design principles behind morphic. When Squeak needed a new user interface environment, John, who had left Sun to join Alan Kay's research group, wrote a completely new version of morphic in Smalltalk. While the details differ, Squeak version retains the spirit and feel of the original morphic, and it is important to acknowledge the debt it owes to the Self project.

Morphic versus MVC

How does morphic differ from the traditional Smalltalk Model-View-Controller (MVC) framework? One difference is that a morph combines the roles of the controller and view objects by handling both user input and display. This design arose from the desire to simplify and from the observation that many view and controller classes were so inter-dependent that they had to be used as an inseparable pair. Many morphs are stand-alone graphical objects that need no model, and some morphs are their own model. For example, a StringMorph holds onto a string rather than a StringModel. However, morphic also supports MVC's ability to have multiple views on the same model, since its browser and other programming tools do precisely that.

Morphic also differs from MVC in its liveness goal In MVC, only one view (i.e., window) is in control at any given time. Only that view can draw on the display, and it must only draw within its own bounds. If it displays anything outside those bounds, say by popping up a menu or scroll bar, then it is must saving and restoring the display pixels below the popped-up object. This display management design is more efficient than morphic's incremental redisplay mechanism since nothing behind the front most window is ever redrawn while that window retains control, which made it an excellent choice for relatively slow machines for which it was developed. However, the MVC design makes it hard to support liveness because there's no easy way for multiple live views to update the screen without drawing all over each other. Morphic's centralized damage reporting and incremental screen updating makes it much easier to support liveness.

Morphic's concreteness is also a departure from MVC. In MVC, feedback for moving or resizing a window is provided as a hollow rectangle, as opposed to a solid object. Again, this is more efficient (only a few pixels must be updated as the feedback rectangle is dragged around, and no view display code must be run) and a good choice for a slower machine.

The Future of Morphic

What lies ahead for Morphic? The Squeak system evolves so rapidly that it is difficult to see very far ahead, but several directions are worth mentioning. First, morphic badly needs an overhaul in its handling of rotation and scaling, features that were retro-fitted into it long after the initial design and implementation were done. The original design decision to have a uniform, global coordinate system should probably be reversed; each morph would then provide the coordinate system for its submorphs with optional rotation and scaling.

The Self version of morphic supported multiple users working together in a large, flat space called "Kansas". From the beginning, it was planned to add this capability to Squeak morphic, but aside from an early experiment called "telemorphic," not much was done. Recently, however, interest in this area has revived, and it may soon be possible to share morphic across the internet.

Efforts are also underway to support hardware acceleration of 3-D, and to allow external software packages such as MPEG movie players to display as morphs. These goals require that morphic share the screen with external agents. Finally, as 3-D performance improves, morphic may completely integrate the 3-D and 2-D worlds. Instead of the 3-D world being displayed within a 2-D morph, today's morphs could become just some unusually flat objects in a 3-D environment.

Further Reading

The following two articles discuss an earlier version of morphic that was part

design issues and cite previous work that influenced the design of morphic. The first paper also describes implementation techniques, while the second focuses on morphic's role in creating a programming experience that reinforces Self's prototype-based object model.

Maloney, J. and Smith, R., "Directness and Liveness in the Morphic User Interface Construction Environment," *UIST '95*, pp. 21-28, November 1995.

Smith, R., Maloney, J., and Ungar, D., "The Self-4.0 User Interface: Manifesting the System-wide Vision of Concreteness, Uniformity, and Flexibility," *OOPSLA '95*, pp. 47-60, October 1995.