Directness and Liveness in the Morphic User Interface Construction Environment

John H. Maloney
Apple Computer, Inc.
1 Infinite Loop, M/S 301-3E
Cupertino, CA 95014 USA
+1-408-974-7293
J.Maloney@eWorld.com

Randall B. Smith
Sun Microsystems Laboratories
2550 Garcia Avenue, MTV 29-116
Mountain View, CA 94043 USA
+1-415-336-2620
Randall.Smith@Sun.com

ABSTRACT
Morphic is a user interface construction environment that strives to embody directness and liveness. *Directness* means a user interface designer can initiate the process of examining or changing the attributes, structure, and behavior of user interface components by pointing at their graphical representations directly. *Liveness* means the user interface is always active and reactive—objects respond to user actions, animations run, layout happens, and information displays update continuously. Four implementation techniques work together to support directness and liveness in Morphic: structural reification, layout reification, ubiquitous animation, and live editing.

KEYWORDS: User interface frameworks, user interface construction, directness, liveness, direct manipulation, animation, structural reification, automatic layout, live editing.

INTRODUCTION
Creating a good user interface is an iterative process. Streamlining this process enables the user interface designer to try more alternatives in search of the best solution. *Directness* means a user interface designer can initiate the process of examining or changing the attributes, structure, and behavior of user interface components by pointing at their graphical representations directly, as opposed to navigating through an alternate representation. *Liveness* means the user interface is always active and reactive—objects respond to user actions, animations run, layout happens, and information displays are updated continuously. Directness and liveness are properties of the physical world: to examine and change a physical object, you manipulate it directly while the laws of physics continue to operate. In a user interface construction environment, directness and liveness reduce iteration time. They also decrease cognitive load by not forcing the designer to correlate graphical components of the interface with their alternate representations (directness) or to switch between run and edit modes (liveness).

UIST 95 Pittsburgh PA USA
© 1995 ACM 0-89791-709-x/95/11..$3.50

Morphic is a user interface construction environment that strives to embody the principles of directness and liveness [10]. Morphic is based on general graphical objects known as *morphs*, from the Greek for "shape" or "physical form." Morphic allows user interfaces and their components to be assembled, disassembled, and rearranged via direct manipulation. It supports interactive automatic layout, animation, and multiple users working simultaneously in a large, virtual space (like Shared ARK [11]).

Morphic draws many ideas from earlier work, although it attempts to go beyond previous systems in consciously harnessing and integrating these ideas in the service of directness and liveness. The section on related work acknowledges some of Morphic's intellectual debt.

Directness and liveness in Morphic are supported by four implementation techniques:
- structural reification (supports directness),
- layout reification (supports directness and liveness),
- ubiquitous animation (supports liveness), and
- live editing (supports directness and liveness).

The remainder of this paper will describe these techniques and how they each contribute to directness and liveness in Morphic.

STRUCTURAL REIFICATION
Complex morphs are constructed by composition (Figure 1). Any morph can be made into a composite morph by attaching other morphs to it as submorphs. A composite morph behaves like a single object: when it is moved, drawn, copied, or deleted, all its submorphs (and their submorphs, recursively) are moved, drawn, copied, or deleted as well. The submorphs of a composite morph are drawn in front of their parent morph and, by default, are given a chance to handle user input events, such as mouse button presses, before their parent. In short, submorphs act as if they were glued onto the face of their parent morph. The submorph structure forms a tree in which every node is a concrete morph and any morph can be a root, leaf, or inner node. Thus, if a composite morph is disassembled,

every part is a visible, manipulable morph. This tangibility of the composite structure is called *structural reification*, and it enables directness in structural manipulation.

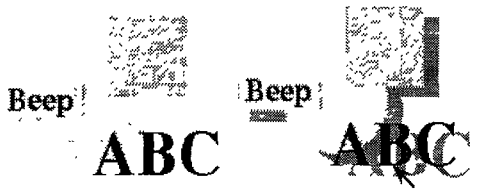

Figure 1. A composite morph (the circle) with three submorphs: a rectangle, a label, and a button. The button itself is a composite morph consisting of a basic button plus a label. A copy of this composite is being moved; the translucent drop shadow indicates that it is temporarily lifted out of the world. Morphs can have non-rectangular boundaries with holes. A composite morph can be grabbed by any part that is not mouse-sensitive; in this case, the copy was grasped by its label submorph.

Many of the components in the Morphic library are composite morphs, allowing them to be modified by direct manipulation. For example, an iconic button could be created by removing the label from a labeled button and replacing it with an arbitrary morph, even a running movie or animation. A menu morph can be "pinned down" (made persistent) and customized by adding, removing, and reordering its buttons. The most extreme example of this composite component approach is an experimental morph-based document editor, in which any "character" in the document can be an arbitrary morph.

Applications constructed with Morphic are just composite morphs that can be built and modified by direct manipulation (Figure 2). Many applications combine morphs that embody the information content of the application (e.g., a morph showing a graph of recent stock prices) with morphs that control this information content (e.g., a slider to control how many days' worth of data is displayed). Morphic allows the user to establish connections between controls and their target morphs by direct manipulation. These connections are stored internally as direct object references: each control morph has a pointer to its target morph. While several alternative reference schemes were considered, including a name-based and several path-based schemes, the direct reference scheme was chosen because it was the most robust in the face of structural changes. (The disadvantage of direct references is that the copy operation must update the copy's intra-composite references.) Thus, the user can arrange the morphs of an application in completely different configurations, possibly inserting or removing new layers of submorphs. This flexibility supports directness and, since controls continue to work when they have been removed from their application component, it supports liveness as well.

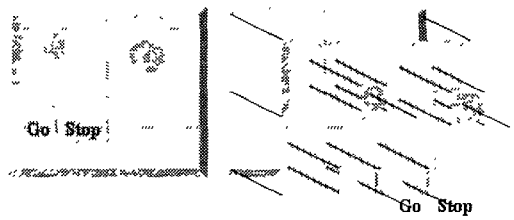

Figure 2. A very simple application (an ideal gas simulation) and an exploded view showing its submorph structure. The exploded view reveals that the highlights on the atoms are just small, light-colored circles. The exploded view is itself a morph constructed by a simple program written by one of the authors.

Structural reification is essential for directness. It allows every graphical object to be examined, manipulated, and disassembled by direct manipulation. The same structuring mechanism is used at all scales, from small components, such as labeled buttons, up to complete applications. Furthermore, using morphs themselves as the structuring mechanism yields a uniformity that allows morphs to be combined freely, even in ways unforeseen by their creators. The next section will show how structural reification can be extended to encompass the placement and sizing of user interface components.

LAYOUT REIFICATION

Automatic layout of user interface components frees the designer from many tedious details of sizing and placement. Without automatic layout, for example, a designer might have to manually adjust the size of a button to accommodate a larger label, and then move several adjacent buttons to make space. Morphic supports automatic layout via *layout morphs*, which adjust the size and placement of their submorphs according to the amount of space available. Just as morphs themselves reify structure, layout morphs reify layout policy, making it something that can be manipulated directly. This *layout reification* supports directness and, since automatic layout appears to operate continuously, liveness.

Currently all layout in Morphic is accomplished using just two types of layout morphs: row morphs and column morphs (Figure 3). A row morph packs its submorphs in a tight horizontal row with no overlaps, while a column morph does the analogous packing vertically. A justification parameter controls placement in the secondary dimension; for example, the tops, bottoms, or centers of a row's submorphs can be aligned with the top, bottom, or center of the row.

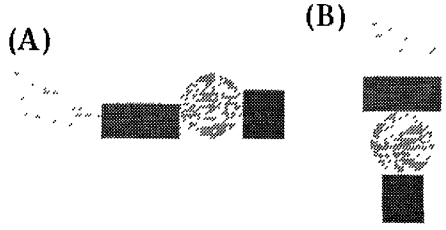

Figure 3. A row morph and column morph. The row is bottom justified while the column is center justified. Both morphs are shrinkwrap, meaning that they shrink to the minimum size required to contain their submorphs. In this example, the row and column have been given a non-zero border width, so their submorphs are slightly inset.

Layout Specification

Row and column morphs, in addition to aligning their submorphs horizontally or vertically, also determine how space should be shared among these submorphs. Space allocation can be thought of as a negotiative process: each layout morph (row or column) finds the best compromise among the space requests of its submorphs. Each layout morph also acts as an intermediary, presenting its owner with a single space request that consolidates the individual space requests of its submorphs.

A morph expresses its space requirements via two attributes. Its *basic minimum size* attribute determines the least amount of space that should be allocated to the morph. Its *resizing* attribute determines how that morph adapts to the availability of extra space. The resizing attribute takes one of three values:

rigid	the morph's size is fixed, regardless of the available space
space fill	the morph expands or shrinks to fill the available space
shrinkwrap	the morph shrinks to smallest size that satisfies the minimum size requirements

The horizontal and vertical dimensions of a morph are completely independent; each dimension has its own minimum size and resizing attributes and each dimension is treated separately by the layout algorithm. The *minimum size* of a morph is computed from its own basic minimum size and the minimum sizes of its submorphs.

Rows and columns use a simple one-dimensional packing algorithm that (1) preserves the size of rigid submorphs, (2) gives each shrinkwrap submorph exactly its minimum space, and (3) gives each space filling submorph at least its minimum space. Any remaining space is divided evenly among all space filling submorphs. This packing algorithm has proven relatively easy to understand and control. Despite its apparent simplicity, nested rows and columns, combined with judicious use of minimum size and resizing attributes, can express a wide variety of layouts (Figure 4). A full programming environment complete with editor, debugger, and hierarchical code browsers has been constructed using these layout facilities, as well as a World Wide Web browser that translates HTML specifications into page layouts built out of rows and columns.

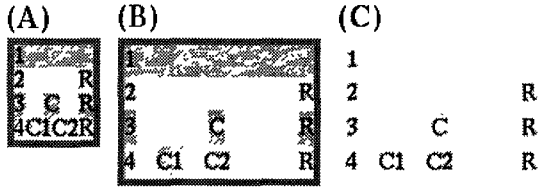

Figure 4: Resizing a column containing four rows from its minimum size (a) to a larger size (b). Nesting has been used to achieve a variety of layout behavior. Here, rows and columns have been given contrasting colors to show the structure, but layout morphs are usually all made the same color (c), so that they are effectively invisible in context yet become visible when removed and placed on a contrasting background. Morphic includes an optimization that suppresses the drawing of nested morphs of the same color; thus, nested layout morphs normally incur no additional drawing costs.

Layout Implementation

Consistent with the principle of liveness, Morphic maintains morph layout continuously. Any operation that might affect layout—such as resizing a morph, adding or removing a submorph, or changing the font size of piece of text—triggers a layout update. One way to achieve this layout update is with a two-pass algorithm: the first pass computes the minimum size requirements of all morphs in the submorph tree from the leaves up to the root; the second pass partitions the available space starting at the root and working back down to the leaves. Unfortunately, the liveness goal may require frequent layout updates, such as after every mouse movement during resizing. This, combined with the fact that composite morphs may include thousands of submorphs, forces the implementation to be more clever. Morphic uses three optimizations to achieve acceptable layout performance.

The first optimization, *deferred layout*, saves the cost of layout when composite morphs are constructed under program control. The idea is to defer layout until the newly created morph is first added to a morph world (a world is just a special composite morph associated with a window); after all, if the morph is not visible, the user shouldn't care if its layout is incorrect. This optimization avoids a potential N^2 cost during morph construction: layout would otherwise be done after adding each submorph, and the cost of each layout operation would be proportional to the total number of morphs that had already been added.

The second optimization is called *pruning*. The idea is to avoid unnecessarily recomputing the layout of submorphs that were laid out during earlier layout computations. This is implemented via a "layoutOkay" flag. When a row or column does layout, it first computes the appropriate size of

each submorph. If the submorph's current size matches this computed size, and if its layoutOkay flag is set, it is simply shifted into position. Otherwise, the recursive call is made to recompute the layout of the submorph. This optimization saves considerable time. For example, consider adding a new row to a column containing a hundred fixed-height rows that are already laid out. Assuming that adding the new row does not affect the width of the column, only the layouts of the column itself and the newly added row are recomputed; pruning saves recomputing the layouts of the hundred existing rows.

The final optimization, *site selection*, attempts to limit the scope of a layout computation to the smallest possible submorph tree. Starting from the site of a change, the system searches up the submorph tree to find a morph whose external layout attributes (size and minimum size) are likely to be stable even when the layout of its submorphs changes (for example, a rigid morph is a good candidate). The layout of this site and its submorph tree is done. If the site's external layout properties do not change, the process terminates. Otherwise, a promising site higher in the submorph tree is chosen, and the process is repeated there. Site selection can save significant time in deeply-nested composite morphs. Without it, even with pruning, the layout of every morph along the path from root to the site of the layout change would be recomputed. Site selection can reduce this to a single, localized recomputation.

UBIQUITOUS ANIMATION

The slogan "ubiquitous animation" encompasses three related ideas. First, Morphic allows morphs to have lightweight autonomous behavior which typically, although not necessarily, appears as animation. For example, a clock might advance the time or a discrete simulation might compute simulation steps autonomously. Second, Morphic allows multiple animations to be active concurrently, even while the system responds to inputs from multiple users; Morphic coordinates display updates among these various activities to keep the display consistent. Finally, Morphic includes a kit of animation behaviors that can be applied to any morph, including motion, scaling, and color change animations. These three facets of ubiquitous animation add a great deal of liveness (and liveliness!) to Morphic (Figure 5).

One important application of autonomous behavior is *observing*. Observing allows an object to respond to changes in other objects without the cooperation of those objects; it is an alternative to notification-based schemes such as Smalltalk-80's dependency mechanism. The Self programming environment relies on observing to update the graphical representation of Self language objects when those objects change. Observing is a polling technique; the observer periodically compares the current observation with the previous observation and performs some action when they differ. Polling means there may be a time lag between a change of state and the display update that reflects this change, but this loose coupling allows rapidly changing values to be observed (sampled) without slowing the computation to the screen update rate. (There are, of course, other ways to achieve such loose coupling.)

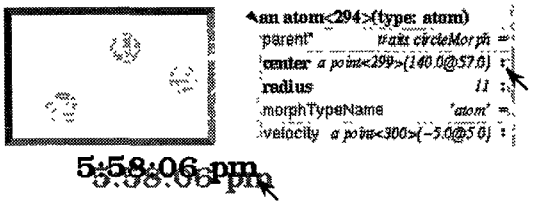

Figure 5. Three simultaneously active morphs: an ideal gas simulation, a digital clock, and a graphical representation of the Self object underlying one of the atoms in the simulation (an outliner). The clock updates every second, the simulation runs continuously, and the outliner uses observing to periodically update its center and velocity slots as the atom bounces around. Any of these morphs continues to operate if it is picked up and moved (e.g., here the clock is being moved) or an external animation is applied to it. Note that two users are working together in this example. Each user has their own cursor and the two users can be active at the same time.

Autonomous Behavior Specification

Although autonomous behavior and other animations are implemented using the same underlying mechanism, they have different purposes and are specified in different ways. The autonomous behavior of a morph is an intrinsic property of that morph. For example, updating the time is central to being a clock morph. Autonomous behavior is defined in the morph itself. Animation, on the other hand, is typically transient and imposed from outside. For example, the Self programming environment gives feedback for certain actions by "wiggling" the relevant morph. Animation is specified by creating a separate *animation activity* object and applying it to the morph to be animated. Animation is orthogonal to autonomous behavior; for example, a clock morph would continue to run even while a motion animation whisked it across the screen.

The autonomous behavior of a morph is defined by its *step* method. For example, a digital clock morph can be created from a label morph by adding a step method that sets the label's contents to a string representing the current time. This behavior is activated by asking the system to send the "step" message to that morph either continuously (every display update cycle) or at periodic intervals (e.g., once per second). Step messages are sent synchronously during the display update cycle. This has the advantage of simplifying synchronization but requires that step methods execute quickly.

An *animation activity* changes some property of its target morph—such as its position, shape, or color—gradually over the course of a number of display cycles (frames). The programmer specifies the beginning and final values of the property to be changed (e.g., the starting and ending

position of a motion animation) and the duration over which the change should occur. The duration can be defined in two ways. *Frame-based* animation lets the programmer control animation smoothness by specifying that the animation should take a given number of frames regardless of the time per frame. *Time-based* animation lets the programmer specify the desired amount of time the animation should take, but the number of intermediate frames depends on the time per frame, which may vary with system load, scene complexity, and other factors. Animations can be paced linearly or slow-in-slow-out. A slow-in-slow-out animation starts slowly, builds to a maximum pace, then decelerates [4]. The system could be extended easily to support other types of pacing.

Animations can be combined into composite animations that execute their components either sequentially or concurrently. An animation can be paused and resumed, restarted, or aborted. Animations can be made to abort automatically if the target morph is grabbed by the user. This facility might be used in a desktop user interface to implemented an abortable delete operation: deleting a file or folder might cause its icon to drift slowly toward the trash can. The user could abort the delete operation by "catching" the icon before it arrived.

Autonomous Behavior Implementation

Morphic uses the well-known damage list technique to support concurrent animation. When a morph is changed in any way that affects its appearance, it is not drawn immediately, but its bounding box is added to the damage list (if it moves, this is done at both its old and its new location). Every display cycle, the damage list is processed, then emptied. All morphs that intersect each damaged rectangle are redrawn in back-to-front order in an off screen buffer and then copied to the screen (i.e., screen updates are double-buffered). The MMM system [1] used a similar technique to coordinate screen updates resulting from the activities of multiple users.

Three refinements of this basic idea improve performance dramatically. First, since the cost of a redisplay cycle depends on the length of the damage list, the list is condensed by merging overlapping damage reports. Second, row and column morphs prune their submorph drawing to the bounds of the damage rectangle being processed. If only a few submorphs of a long row intersect this damage rectangle, then the remaining submorphs are not drawn. (Actually, any composite morph could suppress drawing any of its submorphs that did not intersect the damage rectangle, but rows and columns exploit the fact that their submorphs are packed in a linear, non-overlapping fashion.) Finally, a simple occlusion test suppresses the drawing of morphs that are completely covered by a morph in front of them. (To enable this optimization, the implementor of a morph must assert that it completely fills its bounding rectangle.)

Morphic uses an *activity list* (another old idea) to schedule both autonomous behaviors and animations. The activity list is processed once per display cycle. For each morph being stepped, there is an entry in the activity list (a *periodicStep* activity) that sends "step" to the given morph whenever a certain interval of time has passed since the last time it sent "step" to that morph. A periodicStep activity remains in the activity list until its morph is deleted or its morph's autonomous behavior is explicitly stopped. The other kind of activity list entries, *animation activities* (including composite animations), just perform the next step of their animation every cycle. An animation activity is removed from the activity list automatically when the animation completes.

LIVE EDITING

Many direct manipulation user interface editors make a distinction between *run mode*, during which animations run and buttons and other widgets can be operated, and *edit mode*, during which animation is suspended and buttons and other widgets can be moved or edited but not operated. Morphic avoids this run/edit distinction (sometimes called the "use/mention" distinction [13]). This has several advantages. First, it eliminates the overhead of frequent mode changes. Second, it frees the user of the cognitive burden of remembering the current mode. Finally, in a multi-user system, a run/edit distinction would require users to agree on the desirable mode at any given moment, hampering their ability to work independently.

Modeless editing of potentially mouse-sensitive composite morphs poses a number of interesting problems. The system must provide ways to:

- distinguish editing gestures from operating gestures,
- disambiguate spatial references,
- identify the operands of an operation, and
- manipulate submorphs in place.

This section describes Morphic's solutions to these problems. Directness and liveness issues will be taken up towards the end of the section.

Distinguishing Editing Gestures from Operating Gestures

How does a system without an edit mode distinguish between gestures intended to operate an object from those intended to manipulate it? SUIT [8] uses a special key combination to make this distinction; for example, holding down the SUIT keys while pressing the mouse over a button indicates that the button should be moved, rather than operated. SUIT thus achieves a fair degree of both liveness and directness.

Morphic used a SUIT-like approach initially: special key-mouse combinations were devoted to operations such a move, copy, and delete. However, as this list of operations grew, it became difficult to remember the various key-mouse combinations. Thus, the editing commands were collected into a *meta menu* (Figure 6), popped up by pressing the right mouse button over any morph. (One-button mice could be supported by using a special key-mouse combination to get this menu, as SUIT does.) The meta menu is context sensitive; the operations in it apply to the root of the composite morph under the cursor when the menu is popped up. The meta menu allows the morph to be

duplicated, deleted, resized, disassembled, given a new color, or picked up, even if it is mouse sensitive.

One meta menu command is especially interesting. The "Outliner for Morph..." command summons a graphical interface to the Self language object that implements the morph. This makes it possible to get at the implementation of a user interface component just by pointing at it. It also supports directness in user interface construction: a slot in the underlying Self object can be turned into a button that sends the message associated with that slot. This button can then be used as a component of the user interface. For example, the button could be added to a menu prototype to extend the system.

Figure 6. The meta menu for manipulating a morph. Every menu has an unlabeled bar at the top that allows it to be *pinned* (made persistent) allowing the menu itself to be manipulated. This allows menus to be taken apart into individual (functioning) buttons, reorganized, or otherwise altered.

Disambiguating Spatial References

The user may need to remove a submorph from a composite morph. This raises the problem of disambiguating spatial references to submorphs. That is, since the submorphs of a composite may overlap and nest, pointing at a given screen location is not enough to specify a unique submorph. Morphic uses a technique that might be called "spatial demultiplexing" to disambiguate such references; that is, the stack of submorphs below a single pixel are temporarily represented by a larger area of the screen. One application of this idea is to use a secondary menu (or submenu) to specify the submorph in question. Invoking the "Submorphs" command on a morph pops up a menu listing its submorphs at the spot where the meta menu was invoked, which allows the user to pick the submorph of interest (Figure 7). Another application of this idea is a tool known as the *core sampler*, which will be discussed in a later section.

Identifying Operands

The operations of the meta menu have an implicit operand, the morph on which the menu was invoked. Some operations require additional operands. Morphic uses spatial relationships to specify these operands. (Spatial relationships are very direct!) For example, the "Embed" command makes the morph on which the command was invoked become a submorph of the morph immediately behind (below) it. Drag-and-drop uses a similar spatial relationship to determine two operands: the morph being dropped and the morph immediately below it. The target of control morphs, such as buttons and sliders, is bound by placing the morph over the desired target and invoking the "Set Target" menu command.

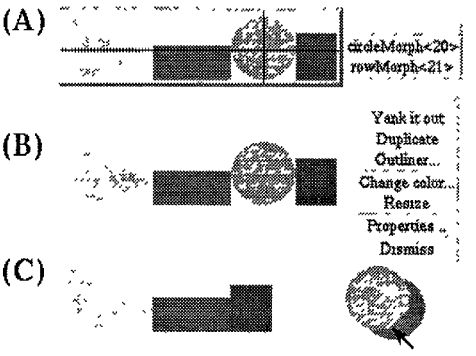

Figure 7. Using the submorph menu to extract a circle from a row. The "Submorphs" command was selected the meta menu, causing the submorph menu to appear (a). The cross hairs indicate the point at which the meta menu was originally invoked. Selecting "circleMorph" in this menu yields a meta menu for the circle (b). Selecting "Yank it out" extracts the circle and attaches it to the cursor (c). Note that the row immediately repacks its remaining submorphs.

Manipulating Submorphs in Place

It is sometimes necessary to modify a submorph of a composite morph. One could extract the submorph from the composite, change it, and replace it, but it is easier to modify submorph in place. One way to do this is through the "Submorphs" command discussed earlier. Selecting a submorph from this menu yields a meta menu that applies to just that submorph.

Another way is to use the core sampler (Figure 8). The core sampler shows a columnar list of submorphs at the screen location in its cross hairs. Each row in this list acts as a proxy for the submorph that it represents. The user can get the meta menu for a submorph by pointing to its proxy. Since resize attributes are so important to layout behavior, proxies make the state of these attributes manifest and manipulable via buttons on the right of each proxy. The user could immediately see, for example, that a row morph was rigid in the horizontal dimension, and could use a popup menu to make it be shrinkwrap instead. Core samplers update their proxy lists continuously as they are moved about the screen (supporting liveness). This allows the user to rapidly probe the submorph structure of any composite morph.

Editing, Liveness, and Directness

The morph manipulation techniques outlined in this section support directness and liveness in several ways. First, these techniques allow editing without introducing a run/edit distinction. Such a distinction would immediately make the system feel less live. Second, the use of spatial

relationships and context sensitive menus enhance directness, because the user operates directly on the graphical object of interest. Where a spatial reference would be ambiguous, the use of spatial demultiplexing in the submorph menu and core sampler afford near-directness by temporarily devoting some screen space to proxies for the objects of interest.

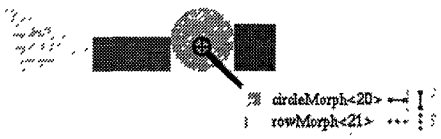

Figure 8. The core sampler with proxies for the two submorphs at its cross hairs, a circle and the row that contains it. The small square on the left side of each proxy shows the color of the associated morph. When the mouse is pressed over this square, a bright-colored frame is temporarily drawn around the submorph to help the user locate the submorph visually. As an accelerator for experienced users, holding the shift key while pressing the mouse over the square extracts the given submorph, while holding the shift key as a morph is dropping onto the square embeds the dropped morph in the given morph.

RELATED WORK

Few of the ideas in Morphic are completely novel, although their combination in a single system is unique and the extent to which directness and liveness are pursued sets it apart. ThingLab [2], and Sketchpad before it [14], pioneered the idea of structural reification and also manifested a certain degree of directness and liveness. Automatic layout based on rows and columns was done in TeX [6] and InterViews [7], although Morphic's space allocation policy is different. Trillium [5], Cardelli's UI builder [3], and the NeXTStep [16] UI builder were early demonstrations of the value of directness in UI construction tools. The Alternate Reality Kit [9] embodied both directness and liveness, but was aimed at constructing physics simulations rather than arbitrary user interfaces. The idea of reserving special gestures to edit a live user interface was used in SUIT [8].

EXPERIENCE

Morphic is implemented in Self, a dynamic, prototype-based object-oriented language [12, 15]. Morphic has been in use for about two years and is the user interface framework of the Self 4.0 programming system. (Self 4.0 is freely available via anonymous ftp from self.smli.com. It requires a Sun SparcStation with a color or gray-scale display.).

Morphic encourages a style in which simple morphs are composed into larger and larger building blocks until a complete user interface is built. How well does this compositional approach scale? The Self programming environment was built this way. Outliners are deeply-nested composite morphs that represent Self objects. An outliner on a large Self object can have thousands of submorphs nested over a dozen levels deep. Such large outliners motivated many of the optimizations discussed in this paper. Thanks to these optimizations, however, performance on a SparcStation 10 is acceptable even for very large composite morphs. Most operations on a composite morph—such as copying, drawing, or updating its layout—are linear in the number of submorphs it has.

Achieving directness is a matter of degree, and Morphic misses the mark in a few ways. Spatial demultiplexing resolves ambiguous submorph references at the cost of a reduction of directness. However, some loss of directness may be unavoidable. That is, given a neatly aligned stack of submorphs with exactly the same shape and size, the quickest way to select one of them seems to be to spread them out spatially, as card players fan out their cards. True, using a menu to disambiguating a submorph reference burdens the user with an extra mouse click, but a single click can select from a large number of submorphs. Another alternative that was considered was to use successive clicks to cycle through the submorphs. However, this would have required many clicks to access a deeply-buried submorph.

A more significant loss of directness arises because, although the user may begin by pointing at the morph of interest, some tasks are accomplished through secondary tools that operate on the morph remotely. For example, changing a morph's color is done through a color changer tool and commonly edited properties are changed through a property sheet. These tools represent areas where the system is not completely polished; with a little work they could be replaced by more direct mechanisms. A deeper philosophical problem arises from the distinction between the morph itself and the programming environment's graphical representation of the Self object that implements that morph (its outliner). If both the morph and its outliner are on the screen, and the user can directly manipulate either, which is the "real" object? Of course, a user interface construction environment that did not allow access to the underlying programming language structures would not face this problem.

CONCLUSIONS

Directness and liveness are design principles. One contribution of Morphic is to demonstrate how these abstract principles can be translated into specific implementation techniques in four general areas. *Structural reification* supports directness by giving the user concrete objects to point at down to a fine granularity. This means that direct manipulation can be used to construct and modify the user interface, and provides a way to get at the underlying implementation of graphical objects. For example, a nice dial morph could be extracted from one application and used to replace a slider in another. *Layout reification* supports directness by embedding layout behavior in morphs, which are visible and directly manipulable. Since layout morphs respond immediately to changes that affect their layout—such as adding, removing, or resizing a submorph,—layout reification also supports a

sense of liveness. Liveness is enhanced by *ubiquitous animation*: autonomous morph behavior, animations, and the activities of multiple-users all proceed concurrently. Finally, Morphic uses a number of interaction techniques to support *live editing*. This frees the user from the unnecessary cognitive burden of a run/edit mode and blurs the boundary between creating and using user interfaces.

The second contribution of Morphic is to validate the hypothesis that directness and liveness are worthy goals. Despite the shortcomings discussed in the previous section, Morphic achieves these goals much more completely than previous systems. The result is a system that is unexpectedly powerful and fun to use. It is difficult for a paper, or even a video, to convey the feeling of engagement and empowerment one gets from using Morphic. Anything you see, you can examine and change. This malleability is the result of striving toward the design principles of directness and liveness.

ACKNOWLEDGEMENTS

Everyone in the Self group has contributed to Morphic, even those who worked primarily on the virtual machine, since the efficiency of the language implementation allowed all of Morphic (down to the XLib calls) to be implemented in Self. Lars Bak designed the outliner view of Self objects. David Ungar implemented a large part of the programming environment. Ole Lehrmann Madsen has been a patient pioneer and insightful critic. Robert Duvall and Brook Conner were courageous enough to teach a course using the system at Brown. Finally, thanks to our loyal users for their feedback and encouragement.

REFERENCES

1. Bier, E., and Freeman, S., "MMM: A User Interface Architecture for Shared Editors on a Single Screen," *UIST '91*, pp. 79-86 (November 1991).

2. Borning, A., "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory," *Trans. on Programming Languages and Systems* 3(4):353-387 (October 1981).

3. Cardelli, L., "Building User Interfaces by Direct Manipulation," *UIST '88*, pp. 152-166 (October 1988).

4. Chang, B. and Ungar, D., "From Cartoons to the User Interface," *UIST '93*, pp. 45-55 (November 1993).

5. Henderson, D., "The Trillium User Interface Design Environment," *CHI '86*, pp. 221-227 (April 1986).

6. Knuth, D., *The TeXbook*, Addison-Wesley (Reading, 1984).

7. Linton, M., Vlissides, J., and Calder, P., "Composing User Interfaces with InterViews," *Computer* 22(2):8-22, (February 1989).

8. Pausch, R., Young, N., and DeLine, R. "Simple User Interface Toolkit (SUIT): The Pascal of User Interface Toolkits," *UIST '91*, pp. 117-125 (November 1991).

9. Smith, R., Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic," *CHI+GI '87*, pp. 61-67 (April 1987).

10. 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*, to appear (October 1995).

11. Smith, R., O'Shea, T., O'Malley, C., Scanlon, E., and Taylor, J., "Preliminary Experiments with a Distributed, Multi-media Problem Solving Environment," in Bowers, J. and Benford, S., ed., *Studies in Computer Supported Cooperative Work: Theory, Practice, and Design*, pp. 31-48, North-Holland (1991).

12. Smith, R. and Ungar, D., "Programming as an Experience: The Inspiration for Self," to appear in *ECOOP '95*.

13. Smith, R., Ungar, D., and Chang, B., "The Use Mention Perspective on Programming for the Interface," in Myers, B., ed., *Languages for Designing User Interfaces*, pp. 79-89, Jones and Bartlett (Boston, 1992).

14. Sutherland, I., "Sketchpad: A Man-Machine Graphical Communication System," *Spring Joint Computer Conference*, pp. 329-345, IFIPS (1963).

15. Ungar, D. and Smith, R., "Self: The Power of Simplicity," *OOPSLA '87*, pp. 227-242, (October 1987). A revised version appeared in *Journal of Lisp and Symbolic Computation* 4(3) (June 1991).

16. Webster, B., *The NeXT Book*, Addison-Wesley (Reading, 1989).