An Introduction to Morphic: The Squeak User Interface Framework

John Maloney

Walt Disney Imagineering

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 boilerplate 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 widgets. For example, a math teacher might 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). To try this for yourself, evaluate “HeadingMorph new openInWorld” in a workspace.

The **HeadingMorph** widget can be implemented in just 5 methods, two for display, two for mouse input, and one for initialization. (The version in the Squeak image includes five additional methods that are not essential.) The process of creating a new morph is easy, because the different aspects of its behavior—its appearance, its response to user inputs, its menu, its drag and drop behavior, and so forth—can be added one at a time. 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.

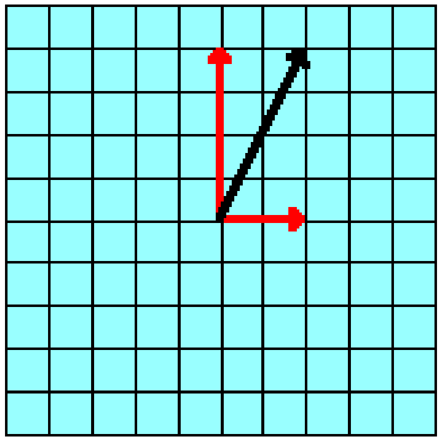

Figure 1: HeadingMorph, a custom widget for showing the x and y components of a vector. This morph can be implemented in just five methods.

At another level, one can work by assembling morphs from the existing library, perhaps using an **AlignmentMorph** or **SystemWindow** 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 Model-View-Controller framework. Another example is **ScorePlayerMorph** (Figure 2), which is just some sliders, buttons, and strings glued together with **AlignmentMorphs**. In all 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. 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, but the environmental support needed for this style of tool construction is currently incomplete.

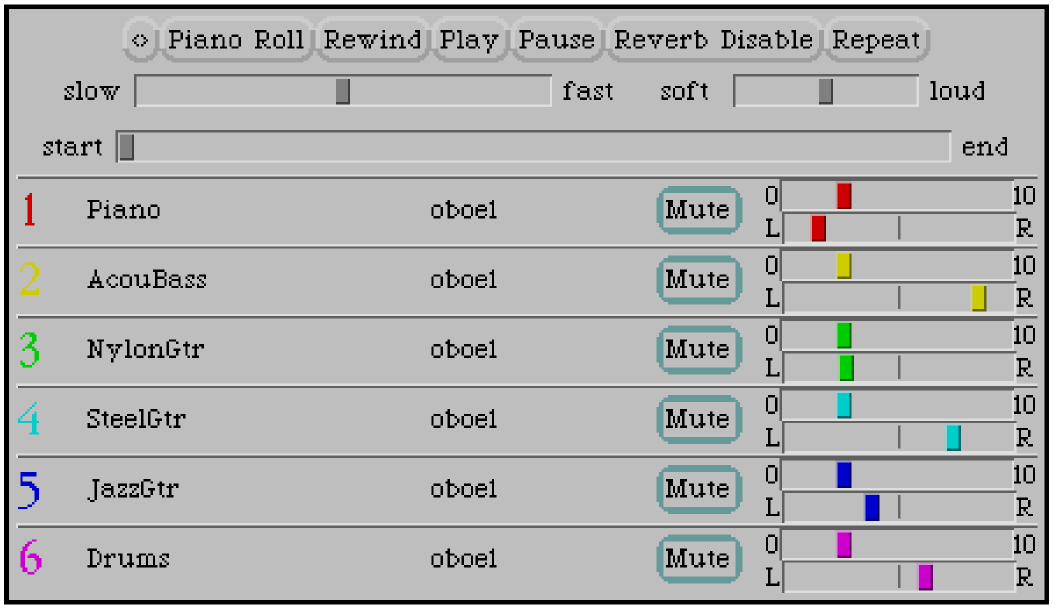

Figure 2: A ScorePlayerMorph, a composite morph assembled from sliders, buttons, strings, and pop-up menu widgets using AlignmentMorphs.

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 (Figure 3). The relative ease with which such viewers and editors can be created is one of the things that sets morphic apart.

At the final level, the Squeak 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 game or a personal information manager. In short, morphic was designed to support the full range of user interface construction activities, from assembling pre-existing widgets to replacing the window system.

Morphic has two faces. The obvious one is the look and feel of the Squeak development environment itself.

This is the face it presents to the user. This chapter focuses on the hidden face of morphic: the one that it presents to the morphic programmer. The first part of this chapter explains the key concepts of morphic from a practical standpoint and works through several examples. The second part describes the deeper principles that guided its design.

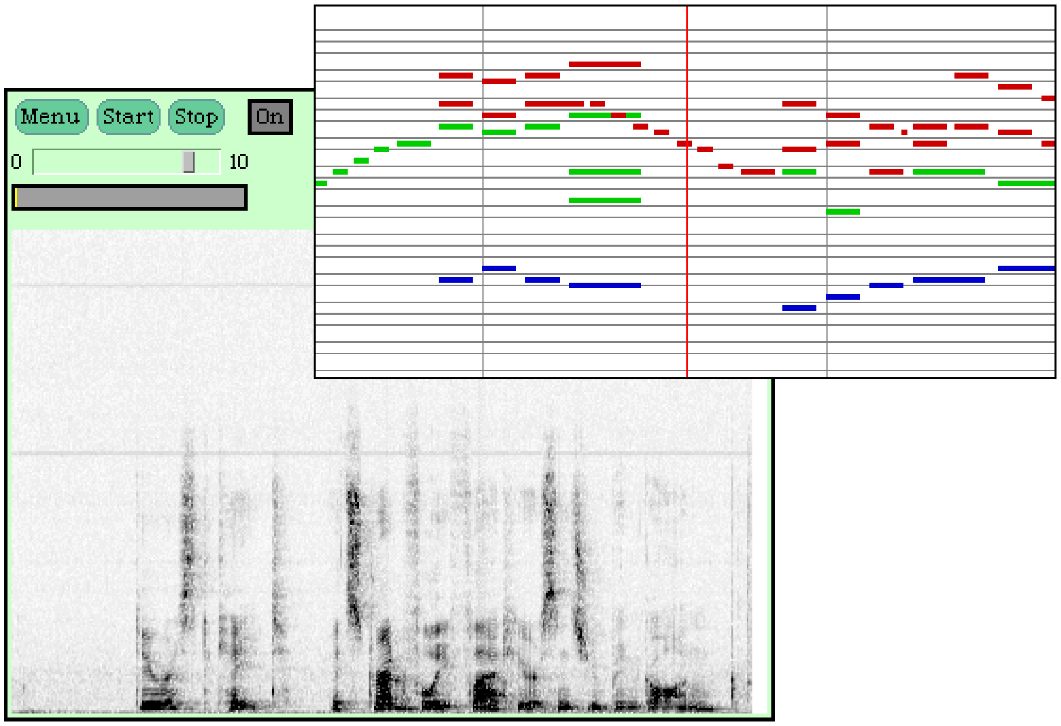

Figure 3: A SpectrumAnalyzerMorph and a PianoRollMorph. These viewers go beyond text by presenting dynamic graphical displays of domain-specific data such as a sound spectra or a musical score. The PianoRollMorph is a simple editor as well as a viewer.

Once the concepts and principles underlying morphic are understood, it becomes easy to create custom widgets, tools, viewers, and editors in morphic—or perhaps even an entirely new windowing system.

Morphs

The central abstraction of morphic is the graphical object or *morph*. Morphic takes its name from the Greek word for "shape" or "form". The reader may have come across the word “morph” in other contexts, where it is used as a verb meaning “to change shape or form 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. Morphs are flat objects that can 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 by providing a drop shadow when a morph is picked up by the user. The drop shadow helps to create 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 through an extended example. We will create a new kind of morph, then add methods to define each aspect of its behavior. While this presentation is not quite detailed enough to be called a tutorial, the code presented is complete, and interested readers are invited to work through the example on their own. The animation, especially, is much more fun to see than to read about.

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 the class **Morph** defines default behavior for everything a morph must do, one can immediately create an instance of **TestMorph** by evaluating:

```
TestMorph new openInWorld
```

The new morph appears as a 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 **Morph**. That's because **Morph** is a concrete class, as opposed to an abstract class (such as **Collection**) that requires that a subclass supply concrete implementations 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** (a subclass of **Canvas**), uses **BitBlt** for drawing, it is possible to create subclasses of **Canvas** that send graphics commands over a socket for rendering on a remote machine or that send Postscript commands to a printer. If color is not available on a device, the canvas can map colors to shades of gray or stipple patterns. Once the **drawOn:** method is written, all these rendering options are available without any additional work; the same **drawOn:** method works with any kind of canvas.

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 command “fillCircle” might get its fill color, border color, and border width parameters from the context object. While the graphics context style of programming might save a bit on parameter passing overhead, it makes many of the parameters that control a given graphic command implicit, making it harder to understand the code. Furthermore, since the context state depends on the history of the computation, different execution paths can 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 to morphs is to browse the class **FormCanvas**, which 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 your **TestMorph** 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 colors 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 its new shape.

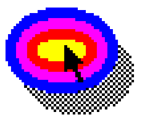

Figure 4: A TestMorph after the drawOn: method has been defined. It has been picked up with the mouse, causing it to cast a shadow.

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 down the ALT key (the command or Apple key on a Macintosh) while clicking the mouse on the morph. Leave the cursor over a halo handle for a few seconds to get a help balloon that explains what the handle does.

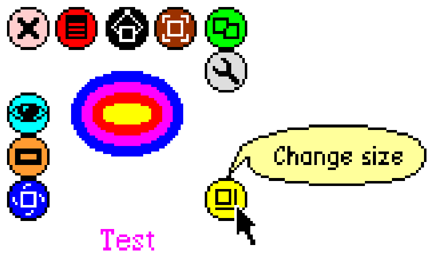

Figure 5: A TestMorph with its halo. The cursor has lingered over a handle for a few seconds, causing the help balloon to appear. Moving the cursor away from the handle will cause the balloon to disappear. Clicking on the background will make the halo itself go away.

Try changing the morph’s size using its yellow (lower right) halo handle. Try changing its color by pressing on the red halo handle 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 “mouse down,” 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. To demonstrate interaction in the context of our example, add the following two methods to **TestMorph**:

```
mouseDown: evt
	self position: self position + (10@0).

handlesMouseDown: evt
	^ true
```

The first method makes the morph jump 10 pixels to the right when it receives a mouse down 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 works for any morph that has its own mouse down behavior.) Another way to handle this problem is to make **TestMorph** reject mouse down events if the shift key is down:

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

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

In addition to 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: aMorph event: evt
```

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 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 sent the message:

```
acceptDroppingMorph: aMorph event: evt
```

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: aMorph event: evt
```

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 several **RectangleMorphs** (using the **new morph...** command 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 picker.

Animation in a user interface can be 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 system query. In the early 1980's, user interface designers realized that the system should not lock the user into 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:

```
initialize
        super initialize.
        angle := 0.

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. Since we've only now added the **initialize** method to set the initial value of angle, any instances of this morph you already have on your screen will have **nil** in their angle instance variable; that will be fixed when you mouse down on the morph.

The liveness of a morph is defined by its **step** method. In this case, the **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 to this morph. If the circle isn't complete, the morph updates its position by moving eight 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. Second, it is good to use an appropriate stepping rate. A clock morph that only displays hours and minutes need not be stepped more often than once a minute (i.e., a step time of 60,000 milliseconds). Finally, the **stepTime** method defines only the minimum desired time between steps; there is no guarantee that the time between steps won't be longer. Morphs that must pace themselves in real-world time can do interpolation based on Squeak's millisecond clock. Squeak's Alice 3-D system does exactly this in order to support time-based animations such as "turn in 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 lines of code.

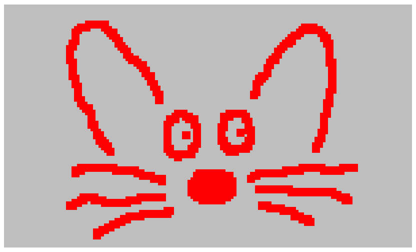

Figure 6: Drawing a picture with PicoPaintMorph, a simple sketch editor.

The first step is, as usual, to create an empty subclass of **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 **Form**), **brush** will be a **Pen** on that **Form,** and **lastMouse** will be used during pen strokes.

The **extent:** method allows the user to make the sketch 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. A **Form** of the new size is created and filled with a background gray color. The contents of the old sketch, if any, are copied into it (using **displayOn:**). If we didn't do this, we'd lose our sketch when the morph was resized. The **nil** test allows this method to be called at initialization time, before the **form** instance variable has been initialized.

To make sure that we start off with a sketch of some default size, we implement the **initialize** method and invoke **extent:** from it :

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

Note that both **extent:** and **initialize** start by invoking the default versions of these methods inherited from **Morph**. These inherited methods handle all the morphic bookkeeping details so that 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 the draw method is trivial:

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

At this point, if you create an instance of **PicoPaintMorph**, it will 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 dragged 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 - bounds origin.
	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 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.

In this case, we don't need to implement the **mouseUp:** method because a default implementation is inherited from **Morph**. In another situation, we could add a **mouseUp:** method to take some final action at the end of the mouse interaction.

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 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 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 process 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 the 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 be independent of the size of the sketch:

```
mouseDown: evt
        brush := Pen newOnForm: form.
        brush roundNib: 3.
        brush color: Color red.
        lastMouse := evt cursorPoint - bounds origin.
        brush drawFrom: lastMouse to: lastMouse.
        self invalidRect:
                ((lastMouse - brush sourceForm extent corner:
                  lastMouse + brush sourceForm extent)
                        translateBy: bounds origin).

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 reports that a portion of the display is no longer valid and must be re-drawn. It takes a 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 reports a slightly larger rectangle than strictly necessary, but it doesn't hurt to redraw a few extra pixels.)

Adding Menu Commands

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. To get started, add the methods:

```
addCustomMenuItems: aCustomMenu hand: aHandMorph
        super addCustomMenuItems: aCustomMenu hand: aHandMorph.
        aCustomMenu add: 'clear' action: #clear.

clear
        form fillColor: Color veryLightGray.
        self changed.
```

The first method adds a new item to the menu that is presented when the red halo handle is pressed. The first line of this method adds the menu items inherited from **Morph**, the next line appends the **clear** command. The **clear** method simply fills the sketch with a neutral color. You can now clear your painting using the red halo handle menu. If you make this menu persistent by selecting **keep this menu up** and keep it handy, you can get to the clear command with a single mouse click.

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.

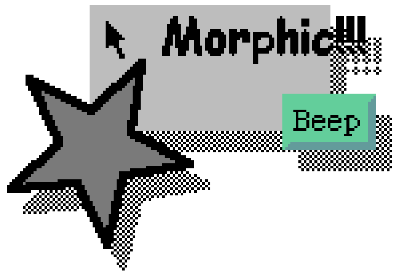

Figure 7: A composite morph created by embedding various morphs in a rectangle morph.

The composite morph is being moved. The embedded morphs stick out past the edge of the rectangle, as 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 objects 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 the morphs themselves. 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 morphs: 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 quandary 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 is 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 have submorphs, be a submorph, or do 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 context. This makes it simpler for objects in a simulation to find out about—and perhaps respond to—their environment. For example, in a billiards simulation, the morph representing the cue stick might search 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 morph 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 it gets to a morph whose owner is a world, hand, or **nil**; 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 submorph 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 positions 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, *layout morphs* actively control the placement and size of their submorphs. The most common 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 does cover a surprisingly wide range of common layout problems. A morphic programmer can also create layout morphs using entirely different layout strategies if necessary, as described later.

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 preceding morph. Submorphs are processed in order: the first submorph is placed at the left end of the row, the second submorph is placed just to the right of the first, and so on. Notice that packing is done only along one dimension—the horizontal dimension in this case. Placement along the other dimension is controlled by the

centering attribute of the **AlignmentMorph**. In the case of a row, the centering attribute 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 stretchable layouts, an **AlignmentMorph** can be designated as space-filling. When there is extra space available during packing, a space-filling **AlignmentMorph** submorph expands to fill it. When there is no extra space, it shrinks to its minimum size. If there are several space-filling morphs in a single row or column, any extra space is divided evenly among them.

Space-filling **AlignmentMorphs** can be used 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 code to create this row is:

```
r := AlignmentMorph newRow color: Color darkGray.
r addMorphBack: (SimpleButtonMorph new label: 'One').
r addMorphBack: (AlignmentMorph newSpacer: Color white).
r addMorphBack: (SimpleButtonMorph new label: 'Two').
r addMorphBack: (AlignmentMorph newSpacer: Color white).
r addMorphBack: (SimpleButtonMorph new label: 'Three').
r inset: 4.
r centering: #center.
r openInWorld
```

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 is centered, and button three gets pushed to the right end.

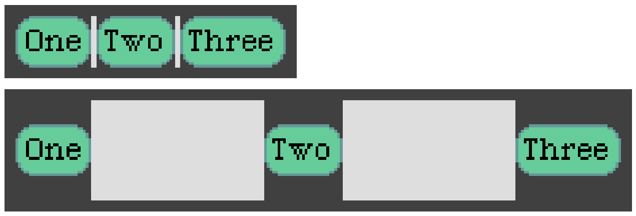

Figure 8: Using space-filling AlignmentMorphs (white) to distribute button morphs evenly within a row. The inset attribute of the row was set to leave a little extra space around its edges. The row is shown at its minimum size and at a larger size. For clarity, the space filling morphs have been made 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 **AlignmentMorph** designated as shrink-wrap grows or contracts to the smallest size that accommodates the space requirements of its submorphs. Here's an example to try:

```
r := AlignmentMorph newRow.
r borderWidth: 1.
r hResizing: #shrinkWrap.
r vResizing: #shrinkWrap.
r addMorph: (StringMorph contents: 'Edit this text!').
r openInWorld
```

Shift click on the label to edit it. Note that the enclosing **AlignmentMorph** grows and shrinks as you change the length of the label string.

Layout Attributes

As we've just seen, **AlignmentMorph** has several 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 attributes can be set to:

rigid	This morph is never resized automatically.
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 larger. Any enclosed space-filling morphs are shrunk as needed.

Custom Layout Policies

AlignmentMorph covers many layout situations, but sometimes a different layout policy is desired. For example, you might wish to create a table of cells for a calendar or spreadsheet, or you might want a layout policy that allowed morphs that didn't fit into one row to move down to the following row, the way words wrap to the next line in many text editors.

In fact, the latter policy is available as an option of **PasteUpMorph**. To try it, create a new **PasteUpMorph** and invoke the **start doing auto-line-layout** command from the **playfield options...** menu. Then drop some morphs into it and see how the layout changes. The layout also adapts as you resize the **PasteUpMorph** with the yellow halo handle.

Implementing custom layout policies, while not difficult, is beyond the scope of this chapter. However, a more advanced morphic programmer might want to study the method **fixLayout** in **PasteUpMorph**. The code to implement wrapping rows of morphs is straightforward. The hook that invokes **fixLayout** is in the method **fullBounds**. The comment in **AlignmentMorph**'s implementation of **fullBounds** explains how the mechanism works. As with many things in morphic, once a custom layout policy is installed, morphic does the rest. The **fixLayout** method will be invoked when morphs are added or removed from the morph, when a submorph changes size, or when the morph itself is resized—in short, anything that could possibly effect the layout.

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; there are 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 it's idle.

.i.Input Processing

Input processing is a matter of 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 front-most morph under the cursor.

Within a composite morph, its front-most submorph is given the first chance to handle an 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 our first attempt at event dispatching, mouse clicks on a submorph were not passed on to its owner, so clicks that hit a button's label were blocked. It is 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 if the cursor moves outside of the scroll bar, and 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. The 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.

Liveness

Liveness 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 a 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 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 might slow things down when building a row or column with lots of submorphs. In fact, since the cost of updating the layout is proportional to the number of morphs already in the row or column, then adding N morphs one at a time and updating the layout after every morph would have a cost proportional to N^2. This cost would mount up fast when building a complex morph like a **ScorePlayerMorph**. To avoid this problem, morphic defers all layout updates until the next display cycle. After all, the user can't see any layout changes until the screen is next repainted. Thus, a program can perform any number of layout-changing operations on a given morph between display cycles and morphic will 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 any aspect appearance (for example, its color), it sends itself the message **changed**, which adds its bounds 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 from the off-screen buffer to the screen is quite fast. The result is the 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 and Directness

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. For example, the page sorter shown in Figure 9 allows the pages of a BookMorph to be re-ordered simply by dragging and dropping thumbnail images of the pages. Since most people have sorted pieces of paper in the physical world, the concreteness of the page sorter makes the process of sorting book pages feel familiar and obvious.

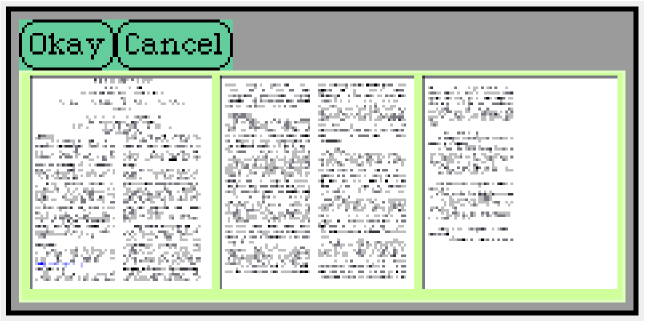

Figure 9: Re-ordering the pages of a BookMorph using a page sorter. Each page is represented by a small thumbnail image. A page is moved by dragging its thumbnail to the desired place in the sequence. The page sorter is handy for sorting "slides" for a Squeak-based presentation.

The user quickly realizes that everything on the screen is a morph that can be touched 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 morphic design principle. Concreteness and directness create a strong sense of confidence and empowerment; users quickly gain the ability to reason about morphs the same 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 user interfaces that show an object being moved only as an outline, morphic always shows 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, pixels are not dribbled onto the screen by some transient process or procedure; rather, the agent that displayed a given pixel is always a morph that is still present and can be investigated and manipulated. Since a morph draws only within its bounds and those bounds are known, it is always possible to find the morph responsible for something drawn on the display by pointing at it. (Of course, in Squeak it is always possible to draw directly on the **Display**, but the concreteness of morphs is so nice that there is high incentive to write code that plays by the morphic rules.)

Halos allow many aspects of a morph—its size, position, rotation, and composite morph structure—to be manipulated directly by dragging handles on the morph itself. This is sometimes called *action-by-contact*. In contrast, some user interfaces require the user to manipulate 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 less common 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. No invisible "container" 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 properties. 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, movies play. Just as in the real world, morphs can 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 supports the use of animation, both for its own sake and to enhance the user experience. For example, if one drops an object on something that doesn't accept it, it can animate smoothly back to its original position to show that the drop was rejected. This animation does not get in the way, because the user can perform other actions while the animation completes.

Liveness also supports a useful technique called *observing*, in which some morph (e.g., an **UpdatingStringMorph**) presents a live display of some value. For example, the following code creates an observer to monitor the amount of free space in the Squeak object memory.

```
spaceWatcher := UpdatingStringMorph new.
spaceWatcher stepTime: 1000.
spaceWatcher target: Smalltalk.
spaceWatcher getSelector: #garbageCollectMost.
spaceWatcher openInWorld.
```

In a notification-based scheme like the Model-View-Controller framework, views watch models that have been carefully instrumented to broadcast change reports to their views. In contrast, observing can watch things that were not designed to be watched. For example, while debugging a memory-hungry multimedia application, one might wish to monitor the total number of bytes used by all graphic objects in memory. While this is not a quantity that is 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 of new mail messages on a 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 there are no display update or layout costs when the value doesn't change. Second, the polling frequency of the observer can be adjusted. Even if it took a full tenth of a second to compute the number of bytes used by all graphic objects in memory, if this computation is done only once a 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 computation to the screen update rate.

A programming environment for children built using morphic shows several examples of liveness (Figure 10). The viewer on the right updates its display of the car's position and heading continuously (an application of observing) as the child manipulates the car. This helps the child connect the numbers representing x and y with the car's physical location. The car can be animated by a script written by the child using commands dragged from the viewer. The script can be changed even as it runs, allowing the child to see the effect of script changes immediately. Individual scripts can be turned on and off independently.

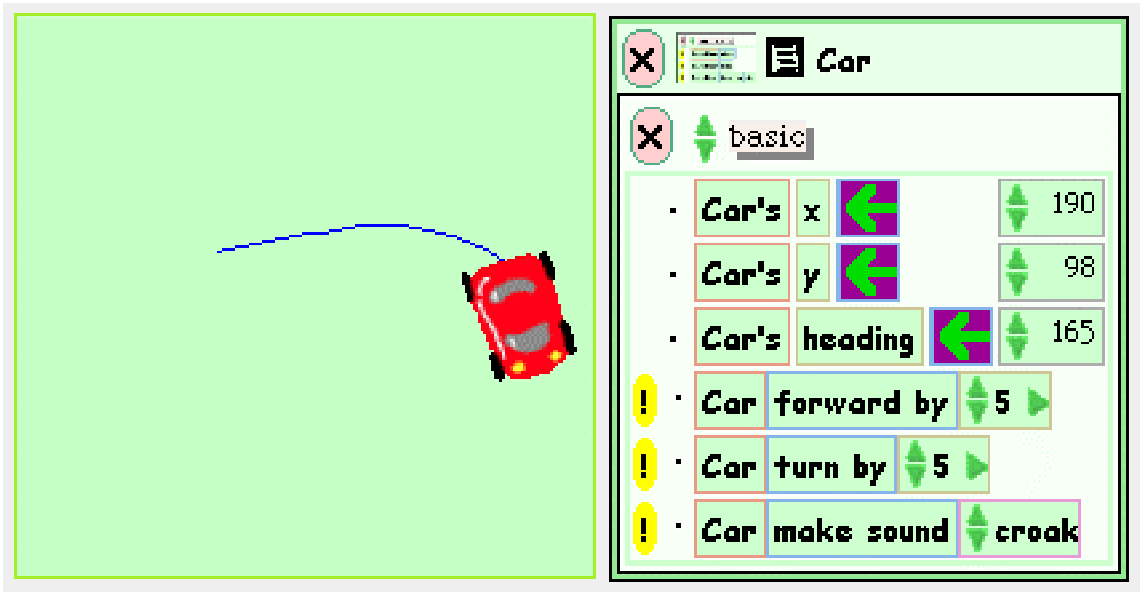

Figure 10: Liveness in a programming environment for children. The car's script runs and the x, y, and heading fields of the viewer update, even as the child writes another script or performs other activities.

The primary mechanism used to achieve liveness is 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. 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 and drop and mouse over behaviors further adds 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 things 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, physical objects obey the same physical laws. We use this uniformity every day to predict how things 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 can 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 **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 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. Randy's previous work with the Alternate Reality Kit and his passion for concreteness and uniformity contributed strongly to morphic's design. For Squeak, morphic was re-written from scratch in Smalltalk. While the details differ, the 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 the Model-View-Controller Framework

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 a desire to simplify and from the observation that most view and controller classes were so interdependent that they had to be used as an inseparable pair.

What about the model? Many morphs are stand-alone graphical objects that need no model, and some morphs are their own model. For example, a **StringMorph** holds its own string, rather than a reference to a potentially shared **StringHolder** model. However, morphic also supports MVC's ability to have multiple views on the same model, using the update mechanism to inform all views of changes to the model. The morphic browser and other programming tools interface to their models exactly the same way their MVC counterparts do.

Morphic also differs from MVC in its liveness goal. In MVC, only one top 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, by popping up a menu or scroll bar for instance, then it must save and restore 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. This was an excellent choice for the relatively slow machines on which MVC was developed. However, the MVC design makes it hard to support liveness because there's no easy way for several live views to interleave their screen updates without drawing over each other. In contrast, Morphic's centralization of damage reporting and incremental screen updating makes liveness easy.

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 screen pixels are updated as the feedback rectangle is dragged around, and no view display code must be run—the right choice for slower machines In fact, morphic itself supports outline-only window dragging and resizing as an option for slow machines.

The Future of Morphic

What lies ahead for morphic? The Squeak system evolves so rapidly that it is likely that any predictions about its future will be old news by the time of publication. Nevertheless, 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.

Morphic is so good at direct manipulation of graphical objects that it seems natural to use morphic itself to assemble tools such as **ScorePlayerMorph** by dragging and dropping components. In fact, this can be done already, although the current tools are rather crude. The real issue is what to do with a morph once it is built. Where is it stored, how is it instantiated, and how are updates and improvements to it propagated, both within an image and to the larger user community? None of these problems is intractable, but they all need to be addressed as part of making morph construction via direct manipulation a practical reality.

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 should soon be possible for several users to share a morphic world across the internet.

Efforts are 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 other agents. As 3-D performance improves, morphic may completely integrate the 3-D and 2-D worlds. Instead of a 3-D world being displayed inside a 2-D morph, today’s 2-D morphs may become just some unusually flat objects in a 3-D environment.

Further Reading

The following two articles discuss the earlier version of morphic that was part of the Self project at Sun

Microsystems Laboratories. Both papers discuss 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 an integrated 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.

About the Author

John Maloney is one of the original Squeak team. John's contributions to the effort include the Smalltalk-to-C translator (key to Squeak's portability), the Macintosh virtual machine, sockets, the sound and music facilities, and the Morphic user-interface framework.

Before he joined the Squeak team, John worked at Sun Microsystems Laboratories, where he and Randy Smith built the first version of morphic for the Self programming environment. John got his Ph.D. from the University of Washington where, as a student of Alan Borning, he developed a fast, constraint-based user interface toolkit written in Smalltalk ("Thinglab II"). Earlier in his graduate school career, he spent a summer in the Smalltalk group at Xerox PARC and eight months working on real-time music accompaniment at Carnegie-Mellon.

In the late 1970's, while John was at M.I.T., he worked as a co-op student at DEC's research lab. It was there that he first saw an early implementation of Smalltalk and fell in love with the language. He still has a faded, illicit photocopy of "Dreams and Schemes," a draft of what eventually became the Smalltalk Blue Book.

John loves Renaissance music. He plays recorders, sings with the Stanford Early Music Singers, and for the past several years has been learning to play the sackbut, an early version of the slide trombone.