Morphic Book

January 23, 2019

Contents

1 Introduction

Morphic

Morphic is the name given to Cuis graphical interface. Morphic is written in Smalltalk, so it is fully portable between operating systems. As a consequence, Cuis looks exactly the same on Unix, MacOS and Windows. What distinguishes Morphic from most other user interface toolkits is that it does not have separate modes for composing and running the interface: all the graphical elements can be assembled and disassembled by the user, at any time. (We thank Hilaire Fernandes for permission to base this chapter on his original article in French.)

2 The history of Morphic

The history of Morphic

Morphic was developed by John Maloney and Randy Smith for the Self programming language, starting around 1993. Maloney later wrote a new version of Morphic for Squeak, but the basic ideas behind the Self version are still alive and well in Pharo Morphic: directness and liveness. Directness means that the shapes on the screen are objects that can be examined or changed directly, that is, by clicking on them using a mouse. Liveness means that the user interface is always able to respond to user actions: information on the screen is continuously updated as the world that it describes changes. A simple example of this is that you can detach a menu item and keep it as a button.
Bring up the World Menu and meta-click once on it to bring up its morphic halo, then meta-click again on a menu item you want to detach, to bring up that item's halo. (Recall that you should set halosEnabled in the Preferences browser.) Now drag that item elsewhere on the screen by grabbing the black handle (see Figure 13.1), as shown in Figure 13.2.

All of the objects that you see on the screen when you run Pharo are Morphs, that is, they are instances of subclasses of class Morph . Morph itself is a large class with many methods; this makes it possible for subclasses to implement interesting behaviour with little code. You can create a morph to represent any object, although how good a representation you get depends on the object!

To create a morph to represent a string object, execute the following code:

```
(StringMorph contents:'Morph') openInHand
```

This creates a Morph to represent the string 'Morph' , and then opens it (that is, displays it) in the world, which is the name that Pharo gives to the screen. You should obtain a graphical element (a Morph), which you can manipulate by meta-clicking.

Of course, it is possible to define morphs that are more interesting graphical representations than the one that you have just seen. The method asMorph has a default implementation in class Object class that just creates a String-Morph . So, for example, Color tan asMorph returns a StringMorph labeled with the result of Color tan printString . Let's change this so that we get a coloured rectangle instead.

```
RectangleLikeMorph new color: Color blue; openInWorld
```

3 Manipulating morphs

Manipulating morphs

Morphs are objects, so we can manipulate them like any other object in Pharo: by sending messages, we can change their properties, create new subclasses of Morph, and so on.
Every morph, even if it is not currently open on the screen, has a position and a size. For convenience, all morphs are considered to occupy a rectangular region of the screen; if they are irregularly shaped, their position and size are those of the smallest rectangular box that surrounds them, which is known as the morph's bounding box, or just its bounds. The position method returns a Point that describes the location of the morph's upper left corner (or the upper left corner of its bounding box). The origin of the coordinate system is the screen's upper left corner, with y coordinates increasing down the screen and x coordinates increasing to the right. The extent method also returns a point, but this point specifies the width and height of the morph rather than a location.

```
joe := RectangleLikeMorph new color: Color blue.
joe openInWorld.
bill := RectangleLikeMorph new color: Color red.
bill openInWorld.
```

Then type joe position and then Print it. To move joe, execute repeatedly:

```
joe morphPosition: joe morphPosition + (10@3)
```

It is possible to do a similar thing with size. joe extent answers joe's size; to have joe grow, execute:

```
joe morphExtent: (joe morphExtent * 1.1)
```

To change the color of a morph, send it the color: message with the desired Color object as argument, for instance,

```
joe color: Color orange
```

. To add transparency, try

```
joe color: (Color orange alpha: 0.5)
```

.
To make bill follow joe, you can repeatedly execute this code:

```
bill morphPosition: (joe morphPosition + (100@0))
```

.
If you move joe using the mouse and then execute this code, bill will move so
that it is 100 pixels to the right of joe.

4 Composing morphs

Composing morphs

One way of creating new graphical representations is by placing one morph inside another. This is called composition; morphs can be composed to any depth.
You can place a morph inside another by sending the message `addMorph:` to the container morph.
Try adding a morph to another one:

```
ellipse := EllipseMorph new color: Color yellow.
joe addMorph: ellipse
```

.

If you now try to grab the balloon with the mouse, you will find that you actually grab joe, and the two morphs move together: the balloon is embedded inside joe. It is possible to embed more morphs inside joe.
In addition to doing this programmatically, you can also embed morphs by direct manipulation.

5 Creating and drawing your own morphs

Creating and drawing your own morphs

While it is possible to make many interesting and useful graphical representations by composing morphs, sometimes you will need to create something completely different.
To do this you define a subclass of Morph and override the drawOn: method to change its appearance.
The morphic framework sends the message `drawOn:` to a morph when it needs to redisplay the morph on the screen. The parameter to `drawOn:` is a kind of `MorphicCanvas`; the expected behaviour is that the morph will draw itself on that canvas, inside its bounds. Let's use this knowledge to create a cross-shaped morph.
Using the browser, define a new class CrossMorph inheriting from `Morph`:

```
CrossMorph
```

We can define the `CrossMorph>>drawOn:` method like this:

```
CrossMorph>>drawOn:
```

Sending the `morphBounds` message to a morph answers its bounding box, which is an instance of `Rectangle`. Rectangles understand many messages that create other rectangles of related geometry. Here, we use the `insetBy:` message with a point as its argument to create first a rectangle with reduced height, and then another rectangle with reduced width.

To test your new morph, execute:

```
CrossMorph new openInWorld
```

However, you will notice that the sensitive zone -where you can click to grab the morph- is still the whole bounding box. Let's fix this.
When the Morphic framework needs to find out which Morphs lie under the cursor, it sends the message `morphContainsPoint:` to all the morphs whose bounding boxes lie under the mouse pointer. So, to limit the sensitive zone of the morph to the cross shape, we need to override the `morphContainsPoint:` method.
Define the following method in class `CrossMorph`:

```
CrossMorph2>>morphContainsPoint:
```

This method uses the same logic as `drawOn:`, so we can be confident that the points for which containsPoint: answers true are the same ones that will be colored in by drawOn . Notice how we leverage the `morphContainsPoint:` method in class Rectangle to do the hard work.

Execute the following code:

```
CrossMorph2 new openInWorld; color: (Color blue alpha: 0.4)
```

There are two problems with the code in the two methods above.

The most obvious is that we have duplicated code. This is a cardinal error: if we find that we need to change the way that horizontalBar or verticalBar
are calculated, we are quite likely to forget to change one of the two occur
rences. The solution is to factor out these calculations into two new methods, which we put in the private protocol:

```
CrossMorph3>>horizontalBar
```

```
CrossMorph3>>verticalBar
```

We can then define both drawOn: and morphContainsPoint: using these methods:

```
CrossMorph3>>drawOn:
```

```
CrossMorph3>>morphContainsPoint:
```

This code is much simpler to understand, largely because we have given
meaningful names to the private methods.

```
CrossMorph3 new openInWorld
```

6 Interaction and animation

Interaction and animation

To build live user interfaces using morphs, we need to be able to interact with them using the mouse and keyboard. Moreover, the morphs need to be able respond to user input by changing their appearance and position; - that is, by animating themselves.

Mouse events

When a mouse button is pressed, Morphic sends each morph under the mouse pointer the message `handlesMouseDown:`. If a morph answers true , then Morphic immediately sends it the `mouseButton1Down:localPosition:` message; it also sends the `mouseButton1Up:localPosition:` message when the user releases the mouse button. If all morphs answer false, then Morphic initiates a drag-and-drop operation. As we will discuss below, the `mouseButton1Down:localPosition:` and `mouseButton1Up:localPosition:` messages are sent with an argument - a `MouseEvent` object- that encodes the details of the mouse action.
Let's extend `CrossMorph` to handle mouse events. We start by ensuring that all crossMorphs answer true to the `handlesMouseDown:` message.

Let's extend CrossMorph to handle mouse events. We start by ensuring that all crossMorphs answer true to the handlesMouseDown: message:

```
CrossMorph4>>handlesMouseDown:
```

Suppose that when we click on the cross, we want to change the color of the cross to red, and when we action-click on it, we want to change the color to yellow. This can be accomplished by the mouse down methods as follows:

```
CrossMorph4>>mouseButton1Down:localPosition:
```

```
CrossMorph4>>mouseButton2Down:localPosition:
```

Notice that in addition to changing the color of the morph, this method also sends self `redrawNeeded`. This makes sure that morphic sends drawOn: in a timely fashion.

Open the CrossMorph:

```
CrossMorph4 new openInWorld
```

Note also that once the morph handles mouse events, you can no longer grab it with the mouse and move it. Instead you have to use the halo: meta-click on the morph to make the halo appear and grab either the brown move handle or the black pickup handle at the top of the morph.
The anEvent argument of mouseDown: is an instance of `MouseEvent` , which is a subclass of `MorphicEvent` . MouseEvent defines the redButtonPressed and yellowButtonPressed methods. Browse this

class to see what other methods it provides to interrogate the mouse event.

Keyboard events

To catch keyboard events, we need to take three steps.

1. Give the keyboard focus to a specific morph. For instance, we can give focus to our morph when the mouse is over it.
2. Handle the keyboard event itself with the handleKeystroke: method. This message is sent to the morph that has keyboard focus when the user presses a key.
3. Release the keyboard focus when the mouse is no longer over our morph.

Let's extend CrossMorph so that it reacts to keystrokes. First, we need to arrange to be notified when the mouse is over the morph. This will happen if our morph answers true to the handlesMouseOver: message.

Declare that CrossMorph will react when it is under the mouse pointer.

`CrossMorph5>>handlesMouseOver:`

This message is the equivalent of handlesMouseDown: for the mouse position. When the mouse pointer enters or leaves the morph, the mouseEnter: and mouseLeave: messages are sent to it.

Define two methods so that CrossMorph catches and releases the keyboard focus, and a third method to actually handle the keystrokes.

`CrossMorph5>>mouseEnter:`

`CrossMorph5>>mouseLeave:`

`CrossMorph5>>processKeystroke:localPosition:`

```
CrossMorph5 new openInWorld
```

We have written this method so that you can move the morph using the arrow keys. Note that when the mouse is no longer over the morph, the `processKeystroke:localPosition:` message is not sent, so the morph stops responding to keyboard commands. To discover the key values, you can open a Transcript window and add Transcript show: anEvent keyValue to the handleKeystroke: method.

The anEvent argument of `processKeystroke:localPosition:` is an instance of `KeyboardEvent`, another subclass of `MorphicEvent`. Browse this class to learn more about keyboard events.

Morphic animations

Morphic provides a simple animation system with two main methods: `step` is sent to a morph at regular intervals of time, while `stepTime` specifies the time in milliseconds between steps . stepTime is actually the minimum time between steps . If you ask for a stepTime of 1 ms, don't be surprised if Cuis is too busy to

step your morph that often. In addition, startStepping turns on the stepping mechanism, while stopStepping turns it off again. isStepping can be used to find out whether a morph is currently being stepped.

Make CrossMorph blink by defining these methods as follows:

```
CrossMorph6>>stepTime
```

```
CrossMorph6>>step
```

```
CrossMorph6 new openInWorld; startStepping
```

7 How Morphic works

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.

7.1 The UI loop

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.

7.2 Input Processing

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.

7.3 Liveness

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.

7.4 Layout Updating

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.

7.5 Display Updating

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.

8 Design Principles Behind Morphic

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.

8.1 Concreteness and Directness

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.

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.

8.2 Liveness

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.

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.

8.3 Uniformity

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 M o r p h , 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.