they all belong to the same Self Unix process).

Figure 1: Three users working in the same space. Two of the other users are collaborating, so they have made their windows overlap (right). The third user is working independently in a separate area (left). The radarView in the third user's area shows the surrounding vicinity in miniature, allowing offscreen objects and the screen boundaries of other users to be seen. The radarView is updated open enough to see where

was pressed. To get a morph to modify, evaluate:

```
morph copy
```

This will make an outliner on a new morph. Use the “Show Morph” command on this outliner’s middle-button menu to make the graphic representation of the copy appear.

The “Add Slot” command on the outliner’s middle-button menu can be used to add a data slot to hold the alternate color. Enter the following expression and accept it by clicking on the green (top) button:

```
otherColor <- paint named: 'leaf'
```

The morph’s drawing behavior can be customized by adding the method:

```
baseDrawOn: aCanvas = (
    aCanvas fillArcWithin: baseBounds
            From: 0
            Spanning: 360
            Color: color.
    self)
```

Morphic optimizes shadow drawing for rectangular morphs such as prototypical morph, which draws as simple rectangle. However, this morph is not rectangular. To make its shadow reflect its true shape, the isRectangular behavior

```
keyUp: evt
mouseMove: evt
leftMouseDown: evt
leftDoubleClick: evt
leftMouseUp: evt
middleMouseDown: evt
middleDoubleClick: evt
middleMouseUp: evt
rightMouseDown: evt
rightDoubleClick: evt
rightMouseUp: evt
```

The event is always supplied so that its state can be examined. The default behavior of the **leftMouseDown:** message is to pick up the composite morph containing the morph that gets the event. (That is, the left mouse button generally means “move”.) The default behavior of the **rightMouseDown:** message is to pop up the morph menu (the “blue” menu). The default behavior of the other messages is to return the special **dropThroughMarker** object, indicating that the event is not processed by this morph.

Submorphs of a morph are displayed in front of their owning morph. By default, submorphs are usually given the first opportunity to handle in TDr rating86 ortunindicatn T : The Sthe fvent. hs are u r oingtit a cin ewole in TDr nt. (That Eachy gagely

# 6 Drag and Drop

A morph can perform some action when another morph is dropped onto it and can decide which dropped morphs it

filling morph a minimum size.

## 7.5 Resize Attribute Summary

The resizing behavior of a morph in one dimension is completely independent of its behavior in the other dimension; that is, a morph actually has two independent resizing attributes, one for the horizontal dimension and one for the vertical dimension.

To summarize, the resizing behavior of a morph along a given dimension is controlled nmor k zing ot8t wo ihdimension;[(Morvior ofs n

There are two ways to achieve animation. First, a morph can have lightweight autonomous behavior which typically, although not necessarily, appears as animation. For example, a clock might advance the time or a discrete simulation sequentially or concurrently. In fact, this activity architecture is the basis of all animation in Morphic: an activity called a **periodicStepActivity** is used to implement the stepping facility.

# 9 Other Issues

## 9.1 Local versus Global Coordinates

The position of a morph is defined relative to the position of its owner. This makes it unnecessary to update the positions of all the submorphs when moving a composite morph. However, it also means that morphs with different owners have positions in different coordinate systems. In order to compare the positions of morphs having different owners, it is necessary to use their positions in the world's coordinate system, which are computed by sending the **globalPosition** message to each morph.
haorphi safelyDo: [ ... ]Tj -19.8-2.3 TD -0.004 Tw (aSnchronization sekuor useully cppear atsintermeittnt Sgraphcall gityces  anlhodug

## 9.2 Synchronization

Animation, stepping, and other activities are handled synchronously, as part of the basic user interface loop. Thus, a sequence of actions done by an activity or a **step** method appear to happen atomically; the user never sees the morph in an intermediate state in which some but not all of the actions have taken place. For example, if a morph is removed from one morph and added to another, the user never sees the transient state in which the morph is not in the world at all. Likewise, any layout modifications resulting from user actions—such as adding a new morph to a row—appear to happen atomically; one never sees a partially complete layout.

Often, however, an independent Self thread wishes to manipulate morphs in the user interface. In order to make such actions appear atomic, they should be done under the protection of the UI synchronization semaphore. The pre mekud to be updated at both the old and new size or position.

Typically, the implementor of a morph writes code to send the changed message automatically after updating any slot that affects the morph’s appearance. For example, the **color:** message defined in traits morph sends changed automatically. Likewise, external animation activities report appropriate changes. Thus, the client of a morph usually need not send **changed** explicitly.

Figure 16: Copying a composite morph. First, the submorph structure of the original morph is copied (a). Then, references among the submorphs of the composite updated to mirror those of the original (b).

## 10 Morph Responsibilities