A Text Field Specification

Ted Kaehler

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2010-002

A Text Field Specification

Ted Kaehler

March 2010

Abstract

This active essay describes how to build a text editor from a series of simple rules in the Lesserphic2 LBox system. Each rule is 'live' and was created within the essay. A rule is a sequence of guarded clauses in a scripting language. Each clause is of the form When ... Do ..., a construction that is understandable by end users. Rules are converted to Smalltalk code and runs in the Lesserphic2 LBox (LObject) model in Squeak. Many text editor features are defined by one rule apiece. A text field with word wrap layout and a working text editor are defined in 37 rules.

Introduction

This is a printout of an active essay that describes a text editor. The essay includes both a written narrative and live code. The code is in the form of rules, with each rule shown in a rule editor window. The rules were constructed inside the essay. Once all of the rules have been accepted, the text editor is defined and can run.

This LBox Text Field Specification is one of a series of prototypes by Ted Kaehler and Alan Kay to find an expressive and understandable end user scripting language. We have chosen the example of word wrap and a text editor because non-programmers can easily picture what these must do. The text layout is expressed from the point of view of a single letter.

To do word wrap, each letter follows the letter before it, and it sometimes needs to go down to the next line. Often (but not in this essay) we include "wandering letters" in our layout rules. When each letter moves incrementally, with a communications delay, a very interesting wandering pattern can be seen.

The essay itself is a Dynabook Junior (DBJr) "stack". DBJr is a HyperCard-like application builder with pages and backgrounds. It is implemented in Morphic in Squeak. Each rule editing window is a live object embedded in a page of the stack. The stack was built with drag-and-drop from a parts bin, and using menu commands.

The right hand column of each rule is a series of guarded actions in the form of When ... Do ... . These clauses only execute when the rule in explicitly invoked. The rule 'place' is invoked with (rule tell aLetter to place). Inside the rule, aLetter is bound to all three pseudo-variables, I, me, and my. Method names that have multiple keyword parts (at least one argument in addition to the receiver) are shown in gray bold text. Keyword parts do not have colons. When a 'return' is executed, we exit the rule and ignore subsequent When-Do clauses. See page 3 of the essay for more about how rules work.

Inside Squeak is Lesserphic2 LObject system built by Yoshiki Ohshima. It has a graphical canvas and nested display objects. It runs inside a Morphic window in Squeak. The compiled rules form the methods of a layout mixin (LWordWrapLayout) that allows a simple box containing letters to become a text editor.

We are proud to be able to express word wrap and a text editor in understandable rules. The first working version of this essay was in October 2009.

Building a Text Field

The goal is to create a text field and arrange its letters in an area of the screen. The letters have an order from first to last. We are doing this in the LObject system, also known as Lesserphic2.

The main problem is to do "word wrap" so that each word is entirely on one line. We don't want half the word at the end of one line and the other half on the next line.

In the most general case, each letter can be any costume that has shape, color, and a bounding rectangle. Any costume that has those properties can be placed in text, even if it is not really a letter. A letter object is rendered into the composition area on the screen using the system's normal costume rendering programs.

With such a general notion of a letter, we are freed from dealing with the details of families of fonts, emphasis (bold, italic), construction of a letter of the desired size, or text color. When a glyph arrives at the layout stage, it already has the proper shape, style, size and color.

Letters in a Box

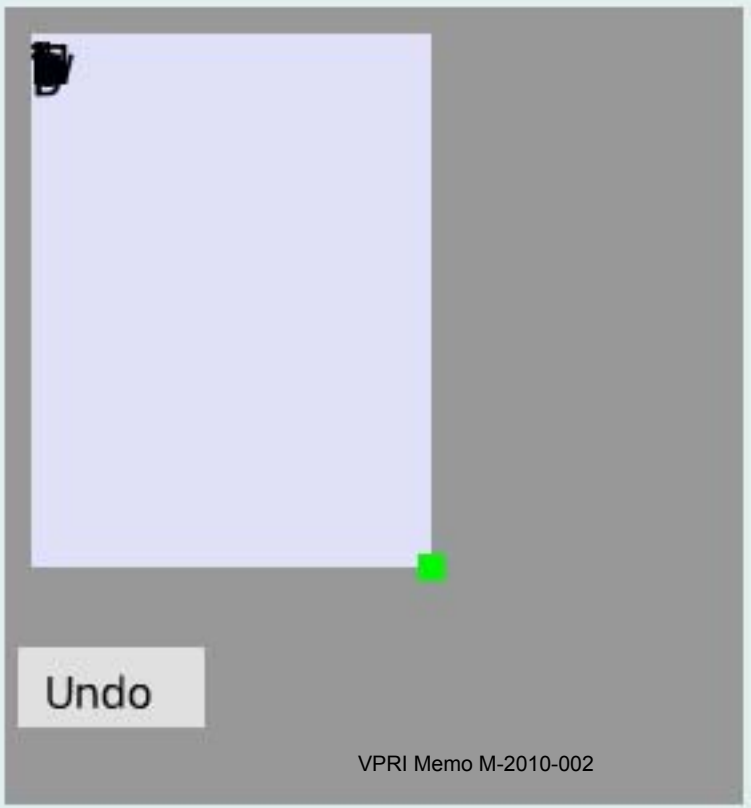
The letters in a text field have an order from first to last.

To see the default situation with no layout rules, press the blue button "Create Text Field". Since there are no rules, all of the letters pile up at the upper left of the field. A mouse click or drag on a letter does nothing.

(To close the demo, Command-click to get a halo on the gray rectangle and Close it using the X in the pink handle on the upper left of the halo.)

When the text extends over

Create Text Field



Text Field Spec

Rules

The behavior of the letters is defined by a set of rules. Each rule is in a rule editor at the right. In the left column, the top pane has the name of the set of rules and the name of this rule. The next two panes are an explanation of the rule. In the right column is the rule itself. The clauses are executed from top to bottom. Some clauses have a guard after the When. If the guard is true, execute the Do part.

"return" means evaluate the expression and hand it back to the place where the rule was called. We exit the rule at the return and do not do the later clauses.

Extremely Simple Layout Methods

Random Layout

As a simple first experiment, we will put each letter in a random place in the text field.

Press **Accept** in the rules for **layout** and **place** at the right.

Press the blue button below. What happens when you move the green square and change the size of the text field?

All in One Line

Now let's redefine **place** to arrange all letters in one long line.

The line is clipped by the edge of the field.

Create Text Field

LWordWrapLayout **layout**	**When** client contents isEmpty not **Do** rule **tell** client contents first **to** 'place'.
Put each character in its proper position in the field.	
Accept	

LWordWrapLayout **place**	**When** I amNil **Do** return me **When Always** **Do** my **positionBecomes** client width atRandom, client height atRandom. rule **tell** my successor **to** 'place'.
Set x,y position of each letter to a random place in the width and height.	
Random placement.	
Accept	

LWordWrapLayout **place**	**When** I amNil **Do** return me **When** my index = 1 **Do** my **position** (client shape **leftAtY** 0)+4 , 4. return rule **tell** my successor **to** 'place'. **When Always** **Do** pred := my predecessor. my **pivot** pred left + pred pivotOffset x + pred width @ pred pivotPosition y. rule **tell** my successor **to** 'place'.
Position me just to the right of my predecessor.	
All in one long line.	
Accept	

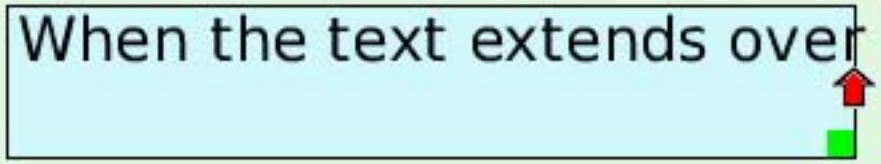
How to Break the Line of Text

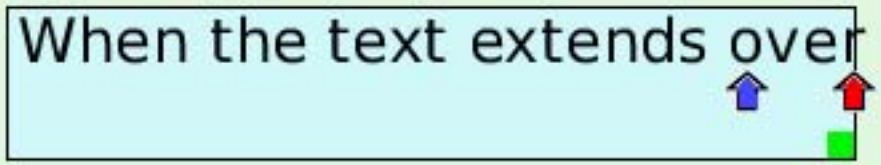
To layout the text in the field, place the next letter just to the right of the previous one.

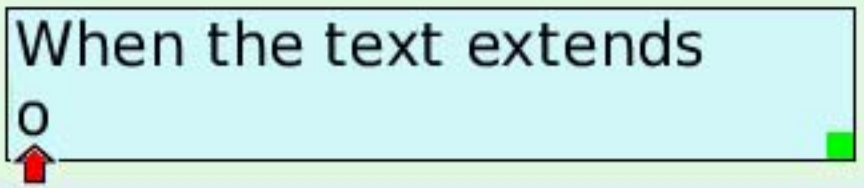
The red arrow shows that the "r" is the letter we just placed.

That letter extends beyond the right margin, so we need to move its entire word to the next line.

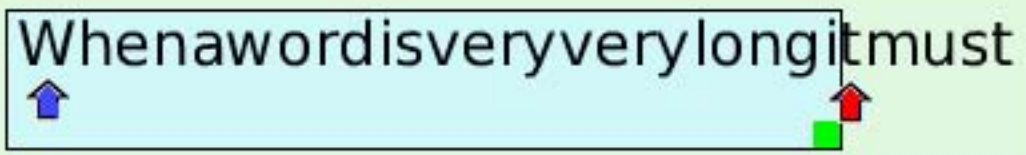
Walk back with the blue arrow until we reach the first letter of the word. Move that letter to the next line. Resume placing letters to the right of the "o", as indicated by the red arrow. Notice that all letters in the word "over" are placed more than once during the layout.

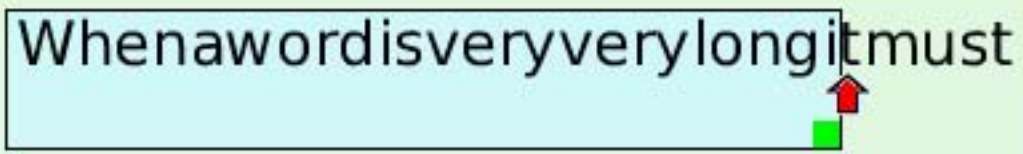






If the walk back (blue arrow) gets all the way to the left margin, a single word covers the entire line. Which letter should be moved to the next line? The original clipped letter (red arrow) is the proper letter to move. A single word that covers the entire line is a special case, and we must test for it.





Wrapping the Text to a New Line

When a line of text is longer than the width of the text field, we want to wrap it to the next line. The goal of text wrapping is to determine where to break the text to start a new line.

Each letter follows its prececessor on the current horizontal line. When a letter hangs over the right margin, its entire word needs to be moved to the next line.

A carriage return causes the next letter to start a new line.

A single word can be wider than entire line. Break it where it touches the right margin. We also need to handle the cases when a letter has no prececessor (it is the first), and has no successor (it is the last).

We start with a general **layout** rule. It places the first letter at the upper left. Then, it the tells the next letter to **place** itself in the field. (Ignore the part about maxHeight and missing- Height for the moment.) When each letter is finished being placed, it must tell its successor to **place**.

Place each letter just to the right of the previous letter on the current line. Then, look for special cases.

If the letter follows a carriage return, move it to the next line (this is done inside **placeIfAfterReturn**).

The letter has the goal of not being clipped by the right margin. When a letter is not white space and finds that it is being clipped, run the **backToWordStart** rule. It looks backwards to find the start of the current word, and moves that letter to the next line.

Press "Accept" for each rule.

Name, Description, Goal	Actions
LWordWrapLayout **layout**	**When** client contents isEmpty not **Do** maxHeight := client contents first shape font ascent. missingHeight := 0. client contents first **pivotBecomes** ((client shape **leftAtY** 0) , maxHeight) + inset. rule **tell** client first successor **to** 'place'. **When Always** **Do** rule **tellLater** rule **to** 'showSelection'.
Move the first letter to the upper left corner. Place the next letter.	
Put each character in its proper position in the field.	
Accept	

LWordWrapLayout **place**	**When** I amNil **Do** return me. **When Always** **Do** pred := my predecessor. my **pivot** pred right + pred pivotOffset x , pred pivot y. **When** rule **placeIfAfterReturn** me **Do** return rule **tell** my successor **to** 'place' **When** rule **isClipped** me **Do** rule **tell** me **to** 'backToWordStart'. **When** (rule **isClipped** me) not **Do** rule **tell** my successor **to** 'place'.
Put the current letter after the previous one. Detect if carriage return, or if the letter is over the right margin.	
Do the normal case when a letter fits on the current line. Look for exceptions.	
Accept	

Noticing Carriage Return and the Right Margin

The rule **placeIfAfterReturn** actually ignores the return character itself. It only takes action when the previous letter is a carriage return. If so, it moves the current letter to the beginning of the next line.

placeIfAfterReturn always returns true or false. This allows it to be used in a guard clause. You can see this in the **place** rule on the previous page. When **placeIfAfterReturn** has moved a letter, it returns true, which signals to go on to the next letter. For all other letters, it returns false, which signals the **place** rule to go further and test whether the current letter is over the right margin.

maxHeight and missingHeight are used to move the line down when a tall letter is in the middle of the line. We won't use these until page 22.

isClipped is the most important rule for specifying word wrap. It returns true if current letter overlaps the right margin. It does this by comparing the letter's right x-value with the margin's x. The margin can be curved, so we ask the text field box for the margin's actual x value at this y. Containers can have irregular shapes, and line lengths can be different.

White space such as a space or a tab are allowed to extend beyond the margin. Return false for white space letters.

LWordWrapLayout **placeIfAfterReturn**	**When** my predecessor shape notNil **and** [my predecessor shape isNewline] **Do** "start of the next line" my **pivotYIncreaseBy** my height. my **pivotLeft** (client shape **leftAtY** my pivotPositionY) + inset x. maxHeight := my shape font ascent. missingHeight := 0. return true.
If my prececessor is a return, move me to the start of the next line.	
Start a new line after a carriage return	**When Always** **Do** return false.
Accept	

LWordWrapLayout **isClipped**	**When** my shape isWhiteSpace **Do** return false.
Answer true if I am not white space and my right is greater than the right margin.	**When Always** **Do** return my right + inset x > (client shape **rightAtY** my pivotPositionY)
Answer whether this letter is over the right margin.	
Accept	

Finding the Start of a Word

We know that the current letter hangs over the right margin. We need to move the entire word to the start of the next line. **backToWordStart** first calls **startOfWord**, which finds the first letter of the current word.

If that letter is already at the start of a line, we should not move it. The line is wider than the field and has no white space in it. The original clipped character should be forced to start a new line.

Otherwise, use the start of the word as the letter to be moved.

Once we have the proper letter in letterToMove, put it at the start of the next line.

LWordWrapLayout **backToWordStart**	**When Always** **Do** letterToMove := self **startOfWord** me. **When** self **isStartOfLine** letterToMove index **Do** "Word takes entire line, break at the clipped character" letterToMove := me. **When Always** **Do** letterToMove **pivotYIncreaseBy** letterToMove height. letterToMove **pivotLeft** (client shape **leftAtY** letterToMove pivotPositionY) + inset x. maxHeight := letterToMove shape font ascent. missingHeight := 0. rule **tell** letterToMove successor **to** 'place'.
Find the start of the current word and move it to the next line. If the word takes an entire line, move clipped letter to next line.	
Accept	

startOfWord travels back along the word to find the first letter. We are looking for a letter that is not white space. If we happen come to the first letter of the text, return it instead.

startOfWord considers just one letter. If that letter is not the start of a word, it calls itself again to consider the preceeding letter.

LWordWrapLayout **startOfWord**	**When** my predecessor isNil **Do** return me. **When** my predecessor shape isWhiteSpace **Do** return me. **When Always** **Do** return rule **startOfWord** my predecessor.
Return the letter at the start of this word.	
Find the beginning of the current word.	
Accept	

Is a Letter at the Start of a Line?

Finally, we need a little test to tell if the current letter is at the start of a line of text. 'me' is the index of a letter.

Press the button to experiment with a field defined by these rules.

Create Text Field

LWordWrapLayout **isStartOfLine**	**When Always** **Do** return (client **at** me) pivot x - inset x <= "left margin" (client shape **leftAtY** (client **at** me) pivotPositionY)
Return true if the letter is at the left margin. Only works on letters that have been placed.	
Accept	

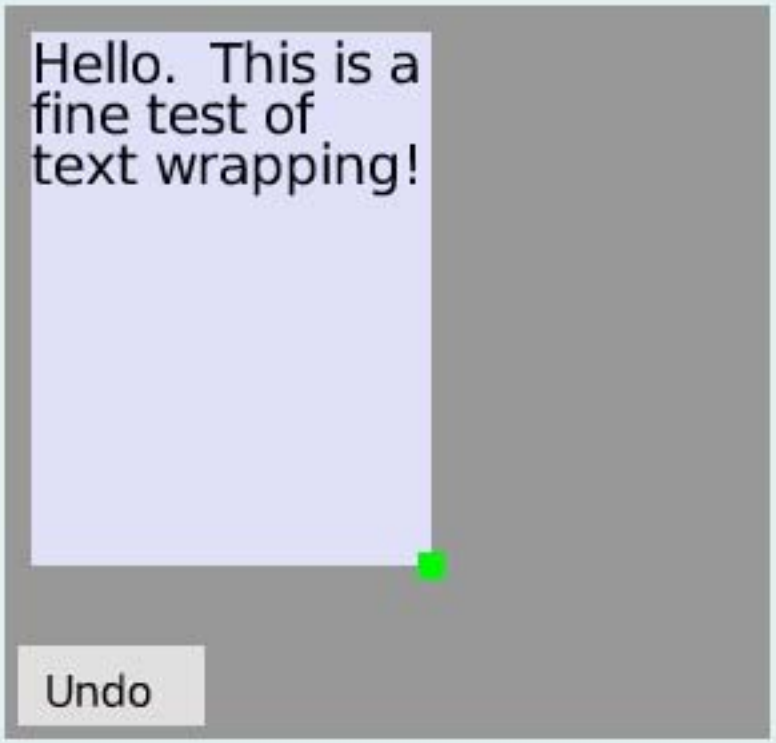
Try the Rules

Create Text Field

When all of the rules have been accepted, press the blue button to see the text field in action. Grab the green square to resize the field. The text will start a new line at a word boundary, as it should.

The **layout** rule is called anytime something in the text field changes. This means that changes in the text contents, the shape of a letter, or the location of a margin will trigger the field to lay itself out again. The field will stay up to date with changes.

These seven rules are enough to specify word wrap in a paragraph of text. They handle several special cases such as an extra long line and carraige return. There are only 17 When clauses in these rules.



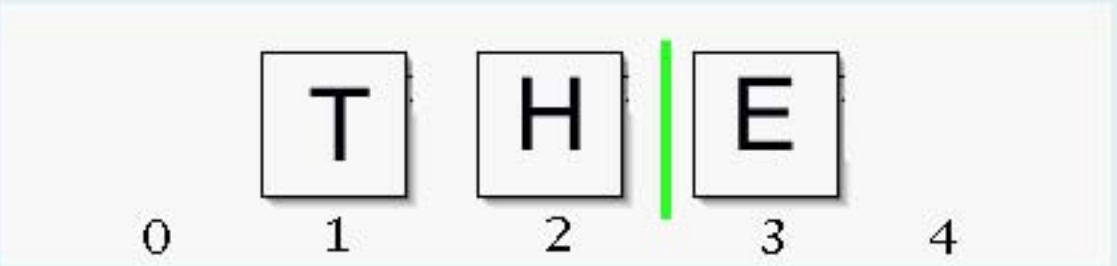
Index Numbers for the Letters

Next, we will implement clicking and dragging in text to select it. A selection starts and ends between letters. An insertion point is a selection with no letters in it. The insertion point can be before the first letter in the field, between any two letters, or after the last letter.

We need a way to store the location of an insertion point, keeping in mind that can be before the first letter or after the last letter. If we simply hold a pointer to a letter and say that the selection begins just before it, we have no easy way to indicate a selection that ends after the last letter.

The **selection** is two numbers, start and end. Start is the number of the letter before the selected text. End is the number of the letter after the selection. The selection is an interval (start to: end) that is non-inclusive of its ends.

The insertion point shown at the right is the selection (2 to: 3). "TH" is the selection (0 to: 3). "THE" is (0 to: 4).



Selecting Text with the Mouse

When the user clicks on a letter, he is trying to select part of the text.

A selection starts between two letters and ends between two letters. The variable **selectionAnchor** holds the index of the first letter selectioned. The selection actually begins on the letter boundary before selectionAnchor.

If the user clicks in the right half of a letter, the selecion should begin at boundary after the letter. We compare the event's position with the point halway across the letter.

A selection can be an insertion point between two letters, or it can span a group of letters. When the user has only clicked, and not moved the mouse, we set the selection to be an insertion point.

If the user clicked on the bare field, call **clickEndOfLine** to put the insertions point at the end of the nearest line of text.

LWordWrapLayout **buttonDown:** anEvent **with**

A click in the left half of a letter means before the letter. In the right half means after. A second click in the same place means select the word.

Set the anchor for a selection in the text. Start with an insertion point.

Accept

```
When Always
    Do client removeProperty 'hideSelection' asSymbol.
When I amNot client
    Do "on a letter"
    letterIP := rule selectionEmpty
        ifTrue [selectionAnchor] ifFalse [nil].
    selectionAnchor := my index.
    (anEvent localPointFor me) x >
        (my width // 2) ifTrue [
            selectionAnchor := selectionAnchor + 1].
    selection := selectionAnchor-1 to selectionAnchor.
    letterIP = selectionAnchor ifTrue ["twice on char"
        anEvent handled rule.
        return rule tell selectionAnchor
            to 'selectWord'].

When I am client
    Do "in open space in the field"
    rule click anEvent endOfLineIn client.

When Always
    Do anEvent handled rule.
    client layoutChanged.
    client removeProperty 'hideSelection' asSymbol.
```

Extending the Selection by Dragging

When the user moves the mouse with the button down, he extends the selection.

If we are in the last half of a letter, use the next letter.

If cursor is after the anchor, select from the anchor to the letter.

If cursor is before the anchor, select from the letter to the anchor. The anchor is the 'pivot point' for the selection.

When the user releases the mouse button, the selection is already correct. We don't need to do anything.

LWordWrapLayout **motion:** anEvent **with**	**When** anEvent buttons > 0 **and** [I **amNot** client] **Do** temp1 := my index. (anEvent **localPointFor** me) x > (my width // 2) **ifTrue** [temp1 := temp1 + 1]. selectionAnchor < temp1 **ifTrue** ["from anchor onward" selection := selectionAnchor - 1 **to** temp1] **ifFalse** ["from temp1 and onward to anchor" selection := temp1-1 **to** selectionAnchor]. client layoutChanged. **When** anEvent buttons > 0 **and** [I **am** client] **Do** temp1 := rule **indexOf** anEvent **in** me. selectionAnchor < temp1 **ifTrue** ["from anchor onward" selection := selectionAnchor-1 **to** temp1+1] **ifFalse** ["from temp1 and onward to anchor" selection := (temp1-1 **max** 0) **to** selectionAnchor]. client layoutChanged. **When Always** **Do** anEvent **handled** rule.
If we are in the last half of a letter, use the next letter. Pivot around the selection anchor.	
Extend the selection when the mouse button is down.	
Accept	

LWordWrapLayout **buttonUp:** anEvent **with**	**When Always** **Do** anEvent **handled** rule.
Accept	

Discovering Which Letter is Near a Click

The user has clicked in the field, but not on any letter. Find out which letter is at the end of the line, and put an insertion point after it. If the click was below the last line, select after the last letter.

LWordWrapLayout **click:** event **endOfLineIn**	**When Always** **Do** index := rule **indexOf** event **in** me. selection := index **to** index+1. selectionAnchor := index+1.
Find the current line, and put an insertion point after the end of it. Assume x is at the end of the line, not before the left margin.	
Accept	

Find the index of the letter clicked on. If below the last line, return the index of the last letter.

Run backwards through the text, finding the last letter of the line where the click occurred.

LWordWrapLayout **indexOf:** event **in**	**When Always** **Do** ourY := (event **localPointFor** me) y. **When** my contents isEmpty **or** [my contents last pivotPositionY < ourY] **Do** "at the end" return my contents size. **When Always** **Do** my contents **reverseDo** [:let | let box lineTop < ourY **ifTrue** ["last letter on this line" return let index]]. return 0
For the event point in the text field, find the index of the letter at that point.	
Accept	

Select Word

The user has double-clicked in a word. Find the start and end of the word and select the word.

Go forward one letter at a time until the end of the word. Then call startOfWord to find the index of the first letter. Select from before the first letter to after the last letter.

LWordWrapLayout **selectWord**	**When** (client contents size >= me **and** [(client **at** me) shape isWhiteSpace not]) **Do** return rule **selectWord** me+1.
Find the first and last letter in the current word.	**When Always** **Do** wordStart := (self **startOfWord** (client contents **atPin** selectionAnchor)) index. selection := wordStart-1 **to** me. client layoutChanged.
Select the word.	
Accept	

Highlight the Selection or Show the Insertion Point

In order to show where the selection is in the text, we put some colored rectangles behind the text.

An LBox actually has three kinds of contents. The letters themselves are in the normal contents. There are other two kinds are "parts". These are things that need to be present, but are not part of the user-defined contents. They include scroll bars, resize boxes, and other controls. One collection of parts is in front of the normal contents and one is behind.

A selection is one, two, or three green rectangles behind the text. There is a rectangle for part of first line, one for all of the fully selected middle lines, and one for part of the last line.

Each selection rectangle is tagged with the property 'selection'.

LWordWrapLayout **installSelection**	**When Always** **Do** client **removeBackPartsSuchThat** [:pp | pp **hasProperty** 'selection' asSymbol].
Create the selection objects and put them into background parts.	**When** (client **valueOfProperty** 'hideSelection' asSymbol) ~~ true **Do** rule selectionEmpty **ifTrue** [rule **installCaret** caret] **ifFalse** [rule **install3Selections** selection].
Remove the old selection, and decide to call for selection or insertion point.	
Accept	

Return a color for the selection rectangles. We use a light green color.

LWordWrapLayout **selectionColor**	**When Always** **Do** return Color **r** 0.258 **g** 1.0 **b** 0.258 "green"
Return the color for the selection rectangles. This is a kind of green.	
Accept	

More Highlighting and Selection

If the selection has zero length, add an insertion point to the background parts of the field.

LWordWrapLayout **installCaret**	**When** caret == nil **Do** rule createCaret. **When Always** **Do** client **addAsBackgroundParts** caret. adj := (font pointSize - caret shape font pointSize) // 2 - caret shape font lineSkip + 1. **When** client contents isEmpty **Do** return caret **pivot** ((client shape **leftAtY** 0), 0) + inset. **When** selection start < client contents size **Do** pt := (client contents **at** selection start + 1) pivot + (-2,adj). **When** selection start >= client contents size **Do** pt := client contents last pivot + (client contents last width-2, adj). **When Always** **Do** caret **pivot** pt.
Add the caret. Compute baseline adjustment. Note when after the last letter. Set position.	
Put the caret object in the right place in the background.	
Accept	

Highlighting a Box of Text

To highlight text that is selected, we place a green box behind the text.

The rule **installSelectionBox** takes a rectangle, creates a box of that size and position, and installs it in the background parts.

LWordWrapLayout **installSelectionBox**	**When Always** **Do** box := LBox **extent** my extent **color** self selectionColor. box **setProperty** 'selection' asSymbol **toValue** true. box **pivotRatio** 0,0. box **pivotPosition** my origin. client **addAsBackgroundParts** box.
Given one rectangle, prepare a box with the selection color and add it to the background behind the letters.	
Install one piece of the selection.	
Accept	

Three Rectangles May be Needed to Cover the Selection

The first line of the selected text may be a partial line, and need a rectangle to cover it. If three or more lines are selected, a middle rectangle is needed. It covers complete lines. The last line may have only a word or two at the beginning selected.

Compute the three rectangles and install them as background parts.

LWordWrapLayout **install3Selections**	**When Always** **Do** letter1 := client **at** selection start+1. letter2 := client **at** selection stop-1. "first line of selection" p1 := letter1 left , letter1 lineTop. x2 := letter1 pivotPositionY = letter2 pivotPositionY **ifTrue** [letter2 right] **ifFalse** [(client shape **rightAtY** letter1 bottom) - inset x]. p2 := p3 := x2 , (p1 y + letter1 shape font lineSkip). rule **installSelectionBox** (p1 **corner** p2).
Create rectangles for selection on start line, selection of whole lines, and selection on end line. Place them in the background. Mark with property selection.	**When** letter2 top > letter1 bottom "middle lines" **Do** p1 := (client shape **leftAtY** letter1 bottom) + inset x , p2 y. p3 := (client shape **rightAtY** letter2 top) - inset x, letter2 lineTop. rule **installSelectionBox** (p1 **corner** p3).
	When letter1 pivotPositionY ~= letter2 pivotPositionY "last line" **Do** p1 := (client shape **leftAtY** letter2 top) + inset x , (p2 y **max** p3 y). p2 := letter2 right @ letter2 pivot y + letter2 shape font descent. rule **installSelectionBox** (p1 **corner** p2).
Accept	

Text Field Spec

Try the Field with Selection and Highlighting

Click the blue button to create a working text field. Click on a letter to test getting an insertion point. Select text to see if dragging out a selection is working. Click in the white space to see if the end of line will be selected.

We have used 11 rule to add text selection.

Accept New Text from the Keyboard

When the user types on the keyboard, **insertChar** receives every letter. We want to insert it into the text, and delete any previously selected text. Create a costume for the letter, give it an event handler, delete the selection, insert the letter into the text, and put the insertion point after it.

LWordWrapLayout **insertChar**	**When** selection start > 0 **Do** ff := (client **at** selection start) shape font. **When** selection start <= 0 **Do** ff := font. **When Always** **Do** glyph := ff **glyphAt** me. letter := LBox **withShape** glyph. rule **installTo** letter. [rule selectionEmpty] **whileFalse** [client **removeAt** selection start+1. selection := selection start **to** selection stop-1]. client **add** letter **afterIndex** selection start. selection := selection start+1 **to** selection start+2.
Create a costume for the letter, give it an event handler, delete the selection, link the letter into the text, and put the insertion after it.	
Catch a keystroke and replace the selection with it.	
Accept	

Move the Inserton Point with Arrow Keys

There are many special keys and control keys that we'd like to use to do things to text. Examples are the arrow keys for moving the cursor and Control-c to copy text.

For each key, we simply add a rule. The name of the rule is 'charTyped' followed by either a letter or a keycode. 29 is the keycode for the right arrow key.

The **right arrow** key moves the insertion point to the right. It also converts a selection to an insertion point. If the insertion point is already after the last letter, do not move it.

LWordWrapLayout **charTyped29**	**When** selection stop <= client contents size **Do** selection := selection stop **to** selection stop + 1. **When** selection stop > client contents size **Do** selection := client contents size **to** client contents size + 1.
Right Arrow. Convert a selection to an insertion point. Don't go beyond the end of the text.	
Move the insertion point one letter to the right.	
Accept	

The **left arrow key** moves the insertion point to the left. It also converts a selection to an insertion point. If the insertion point is already before the first letter, do not move it. The keycode for left arrow is 28.

LWordWrapLayout **charTyped28**	**When** selection start >= 1 **Do** selection := selection start - 1 **to** selection start. **When** selection start < 1 **Do** selection := 0 **to** 1.
Left Arrow. Convert a selection to an insertion point. Don't go beyond the start of the text.	
Move the insertion point one letter to the left.	
Accept	

Down Arrow

Down Arrow moves the insertion point to the next line. It also converts a selection to an insertion point. If the insertion point is already after the last letter, do not move it. (For the moment, move to the beginning of the line, not to the letter just below its previous position.) The keycode for down arrow is 31.

LWordWrapLayout **charTyped31**	**When** selection stop > client contents size **Do** return selection := client contents size **to** client contents size + 1. **When Always** **Do** endPrevLine := (rule **startOfNextLine** selection stop + 1) - 1. selection := endPrevLine **to** endPrevLine+1.
Down Arrow. Convert a selection to an insertion point. Don't go beyond the end of the text.	
Move the insertion point to the next line.	
Accept	

Find the start of the next line. Return the index of this letter if it is at the start of a line. Otherwise ask my successor. If I am at the end, return the index of the last letter.

LWordWrapLayout **startOfNextLine**	**When** client contents size <= me **Do** return client contents size + 1. **When** (rule **isStartOfLine** me) not **Do** "not at left margin" return rule **startOfNextLine** me+1. **When Always** **Do** return me "at start of next line"
Return the index of this letter if it is at the start of a line. Otherwise ask my successor. If I am nil or at the end, return the last letter.	
Find the start of the next line.	
Accept	

Up Arrow

Up Arrow moves the insertion point to the previous line. It also converts a selection to an insertion point. If the insertion point is already before the first letter, do not move it. The keycode for down arrow is 30.

Note that the first call on **startOfLine** moves to the start of the current line, and the second moves to the line above it.

LWordWrapLayout **charTyped30**	**When** selection start < 1 **Do** return selection := 0 **to** 1. **When** self selectionAtEnd **Do** lastOfPrevLine := (self **startOfLine** selection start) - 1. selection := (self **startOfLine** lastOfPrevLine) - 1 **to** (self **startOfLine** lastOfPrevLine). **When** self selectionAtEnd not **Do** lastOfPrevLine := (self **startOfLine** selection start + 1) - 1. selection := (self **startOfLine** lastOfPrevLine) - 1 **to** (self **startOfLine** lastOfPrevLine).
Up Arrow. Convert a selection to an insertion point. Don't go beyond the start of the text.	
Move the insertion point up to the previous line.	
Accept	

Find the start of the current line. Return the index of this letter if it is at the start of a line. Otherwise ask its predecessor. If the index is at the beginning, return 1.

LWordWrapLayout **startOfLine**	**When** 1 > me **Do** return 1. **When** (rule **isStartOfLine** me) not **Do** return rule **startOfLine** me-1. **When Always** **Do** return me. "at start of this line"
Go back letter by letter until one is at the margin.	
Find the index of the letter that is at the start of the current line.	
Accept	

Editing Commands

Every user expects to **copy** text by typing **Control-c** (Command-c on a Macintosh). Here is the rule that does it. When the Control key is down and a letter is typed, the system looks for a method named 'charTypedCMD' followed by either a letter or a keycode. **charTypedCMDc** is called when the user types a 'c' with the Control key down.

Copy the text selection and put it into the world's clipboard.

LWordWrapLayout **charTypedCMDc**	**When** self selectionEmpty not **Do** client worldState **clipboard** ((selection start + 1 **to** selection stop - 1) **collect** [:ii | (client contents **at** ii) copy])
Make sure the selection is not empty.	
Command-c. **Copy** the selection and put it into the world's clipboard.	
Accept	

In a similar way, define **Control-x** to be the **cut command**. First do the copy command, then delete the selection.

LWordWrapLayout **charTypedCMDx**	**When Always** **Do** rule charTypedCMDc. [rule selectionEmpty] **whileFalse** [client **removeAt** selection start+1. selection := selection start **to** selection stop - 1].
First do the copy command, then delete the selection.	
Command-x. **Cut**. Delete the selection and put it into the clipboard.	
Accept	

Paste and Backspace

When the user types **Control-v, paste** the text in the clipboard into the field.

Make sure the clipboard has something in it. Later, we will revise this to make sure the thing in the clipboard meets the requirements to be in text.

Delete the current selection in the text.

For each letter in the selection, copy it and insert it just after the insertion point.

Move the insertion point to after the new letter.

LWordWrapLayout **charTypedCMDv**	**When** client worldState clipboard isNil **Do** return rule. **When Always** **Do** [rule selectionEmpty] **whileFalse** [client **removeAt** selection start+1. selection := selection start **to** selection stop-1]. client worldState clipboard **do** [:lbox | letter := lbox copy. rule **installTo** letter. letter setPivotForGlyph. client **add** letter **afterIndex** selection start. "put insertion point after new text" selection := selection start+1 **to** selection start+2].
Delete the selection and replace it with a copy of the clipboard.	
Command-v. **Paste** text into the field.	
Accept	

When the user types the **backspace** or **delete key**, several thing may happen. '08' is the keycode of the delete key.

If the selection is empty, and there is a letter before it, delete that letter.

If the selection is not empty, simply delete the selection.

Press the button below and try cutting and pasting text.

Create Text Field

LWordWrapLayout **charTyped08**	**When** self selectionEmpty **and** [selection start > 0] **Do** client **removeAt** selection start. selection := selection start - 1 **to** selection start. **When** self selectionEmpty not **Do** [self selectionEmpty] **whileFalse** [client **removeAt** selection start + 1. selection := selection start **to** selection stop - 1].
If insertion point, erase the letter before it. If a selection, erase it.	
Delete or Backspace. Remove the selection.	
Accept	

Tab Stops

A tab is unique. It's width depends on where it is in the line of text. An array of numbers called tabArray holds the x values of tab stops.

When we layout a tab, we need to set its width each time. Run backwards through the tab stops, looking for one just larger than the tab's left edge. Set the tab's width so that the next letter will be at that stop.

LWordWrapLayout **setTabWidth**	**When** my shape ~~ nil **and** [my shape name == #controlHT] **Do** goalX := client width. tabArray **reverseDo** [:tabX | tabX > me left **ifTrue** [goalX := tabX]]. my **shape** (my shape **asTabOfWidth** goalX - my left).
If I am a tab, go backwards through the tab stops, looking for one just larger than my x. Set my width so that the next letter will be at that stop.	
Set the width of a tab character.	
Accept	

Tall Letters

The height of a line is the font height of the first letter. If there is a letter in a larger font in the middle of the line, we need to move the entire line down.

When a taller letter is discovered, run back through all the previous letters in the line and move them down.

If this letter is taller than we have seen (maxHeight + missingHeight), add the extra to missingHeight.

You may have noticed that layout on page 3 has lines that initialize maxHeight and missingHeight.

LWordWrapLayout **tallLetter**	**When Always** **Do** extra := my shape font ascent - (maxHeight + missingHeight). **When** extra > 0 **Do** missingHeight := missingHeight + extra.
If this letter is taller than all we have seen on this line, increase missingHeight.	
Set the variable missingHeight when we see a taller font.	
Accept	

Base Line Adjustment

Run back from the tall letter to the beginning of the line, moving each letter down the page.

LWordWrapLayout **baselineAdjust**	**When** I amNil **or** [missingHeight <= 0] **Do** return self. **When Always** **Do** my **pivotPositionY** my pivotPositionY + missingHeight. **When** self **isStartOfLine** my index **Do** maxHeight := maxHeight + missingHeight. missingHeight := 0. **When** (self **isStartOfLine** my index) not **Do** rule **baselineAdjust** my predecessor.
If missingHeight is > 0, then lower this letter and move back towards the start of line.	
Move letters down when a taller letter is discovered.	
Accept	

Every time we **place** a letter at its final location, we need to check if it is a tab. Since a letter has its x set in three different places, we will wait until we place the next letter to compute a final tab width.

In the **place** rule, the first thing we will do is to **setTabWidth** of the previous letter. This should work for all letters, including the first letter. (If the last letter is a tab, its width will be wrong. But, no letter follows, so it is OK.)

This is the second time we have modified the **place** rule to add more capability. Be sure to click Accept.

LWordWrapLayout **place**	**When** I amNil **Do** return me. **When Always** **Do** pred := my predecessor. rule **setTabWidth** pred. my box **pivot** pred right + pred pivotOffset x @ pred pivot y. self **tallLetter** me. **When** rule **placeIfAfterReturn** me **Do** return rule **tell** my successor **to** 'place'. **When** my shape isWhiteSpace **and** [(self **isClipped** me) not] **Do** rule **tell** me **to** 'baselineAdjust'. **When** rule **isClipped** me **Do** rule **tell** me **to** 'backToWordStart'. **When** (rule **isClipped** me) not **Do** rule **tell** my successor **to** 'place'.
Put the current letter after the previous one. Detect if carriage return, or if the letter is over the right margin.	
Do the normal case when a letter fits on the current line. Look for exceptions.	
Accept	

Changing Emphasis

To make text be **bold**, select some text and press **Control-b**. Or, select the text and choose Bold from the Style Menu. If the beginning of the text is already bold, make it un-bold.

For each letter of the selection, tell its surface (letter shape) to be the same surface asBold.

LWordWrapLayout **charTypedCMDb**	**When** (client **at** selection start + 1) isBold not **Do** return rule **selectionDo** [:let | let beBold] **When** (client **at** selection start + 1) isBold **Do** rule **selectionDo** [:let | let box beNotBold].
To make text be **bold**, select some text and press **Control-b**. If the beginning of the text is already bold, make it un-bold.	
Toggle the bold property of the selection.	
Accept	

To make text be **italic**, select some text and press **Control-i**. Or, select the text and choose Italic from the Style Menu. If the beginning of the text is already italic, make it un-italic.

For each letter of the selection, tell its surface (letter shape) to be the same surface asItalic. A letter can be bold and itialic at the same time.

LWordWrapLayout **charTypedCMDi**	**When** (client **at** selection start + 1) isItalic not **Do** return rule **selectionDo** [:let | let beItalic] **When** (client **at** selection start + 1) isItalic **Do** rule **selectionDo** [:let | let beNotItalic].
To make text be **italic**, select some text and press **Control-i**. If the beginning of the text is already italic, make it un-italic.	
Toggle the italic property of the selection.	
Accept	

Plain Text

When the user types **Control-t** for **plain text**, remove any bold or italic from the selection. For each letter in the selection, replace its graphic with the non-bold, non-itialic version.

LWordWrapLayout **charTypedCMDt**	**When Always** **Do** rule **selectionDo** [:let | let beNormal].
Control-t, plain text. For each letter in the selection, replace its graphic with the non-bold, non-itialic version.	
Force there to be no bold or italic in the selection when the user types Control-t.	
Accept	

Change the Font

Change the font of all letters in the selection. This is invoked from the menu of the text field box. Control-click in the text field to get a halo. Click on the menu icon at the upper left. Click "Choose Font". Pick a font from the list.

In this method, if (I am: nil) is true, it means that the user chose 'leave as it'. Exit in this case.

If ('be the default' = me), the user designated the font of the first letter in the selection to be the default font for this field. What is that used for? If you delete all the text in the field, and then type, the new letters will be in the default font.

LWordWrapLayout **selectionToFont**	**When Always** **Do** client topContainer worldState deleteHalo. **When** I **am** nil **Do** return self. **When** 'be the default' = me **Do** return font := (client **at** selection start + 1) shape font. **When Always** **Do** newFont := LFamily families **at** me. self **selectionDo** [:let | "replace the glyph shapes in place" let **shape** (let shape **ofFont** newFont)]. client layoutChanged.
Change the Font of the selection.	
Change the selection to that font, preserving, size and emphasis.	
Accept	

Increase Font Size

Make all of the letters in the selection be one size larger. The keycode of "plus" is 43.

LWordWrapLayout **charTypedCMD43**	**When Always** **Do** rule **selectionDo** [:letter | letter **increaseFontBy** 1].
Control- +, Control-=. **Increase font size** of the selection. Handles both = and + since the shift key is not tested.	
Accept	

Decrease Font Size

Make all of the letters in the selection be one size smaller. The keycode of "minus" is 45.

LWordWrapLayout **charTypedCMD45**	**When Always** **Do** rule **selectionDo** [:letter | letter **increaseFontBy** -1].
Control-minus. **Decrease font size** of the selection.	
Accept	

Select All Text

Type **Command-a** to select all of the letters in the field.

LWordWrapLayout **charTypedCMDa**	**When Always** **Do** selection := 0 **to** client contents size + 1
Command-a. **Select all text.**	
Accept	

Conclusion

Each rule is translated into the base language of the system. The result runs as fast as any normal text editor. There is no speed penalty for using rules to define the behavior.

Basic layout and word wrap takes 7 rules. Selection, highlighting, typing and clicking in the open field takes 11 rules. Fifteen addtional features take 19 rules. The entire text editor is defined in 37 rules.

Rules of this kind are easy to define. The sequence of clauses with guards is very flexible and expressive. The resulting application runs at full speed, and is understandable by humans.

First working version: October 2009
Current version: March 2010