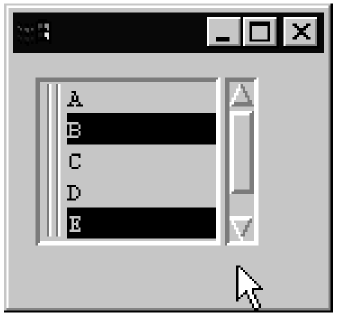
Text lists

The text list vidget displays a scrollable list of lines of text. Here is a text list with the letters A-F:



A text list can be configured as "select one", "select many", or "read only". The list can be scrolled vertically but not horizontally.

The *items* of a text list can be set with `VSetItems()`:

```
VSetItems(vidgets["list1"],
          ["A","B","C","D","E","F"])
```

There is no provision for adjusting the list other than to call `VSetItems()` with a different list of values.

The list can be retrieved with `VGetItems(`*V*`)`.

Text lists, continued

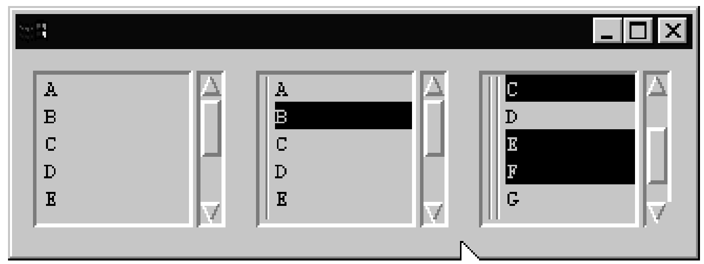
Here is the VIB-generated callback for a text list:

```
procedure list_cb1(vidget, value)
   return
end
```

If the list is single-selection, clicking on an item (e.g., "A") produces a callback with `value` equal to `"A"`.

If the list is multiple-selection, the callback is invoked with a list of the currently selected items, such as `["A"]`, `["B","E"]`, or `[]` (if no items are selected).

The *state* of a list can be retrieved with `VGetState(V)`. The value produced is a list. The first element is the index of the first visible list entry. The following elements are the indices of the selected entries, if any.

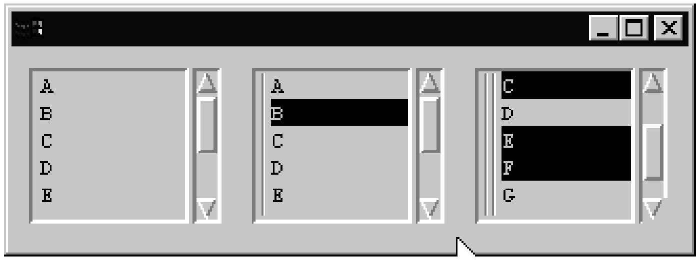


(`textlist2`)

```
Vidget: list2
Value: "B"
State: [1,2]

Vidget: list3
Value: ["C","E","F"]
State: L7:[3,3,5,6]
```

Text lists, continued

For reference:



Here is the pertinent code:

File scope:
```
global vidgets
```

In `main`:

```
every VSetItems(vidgets["list"||(1 to 3)],
            ["A","B","C","D","E","F","G","H"])
```

In `list_cb`:

```
procedure list_cb(vidget, value)
    vidget := vidgets[vidget.id] # REQUIRED!?!
    write("Vidget: ", vidget.id)
    write("Value: ", Image(value,3))
    write("State: ", Image(VGetState(vidget),3))
    write()

    return
end
```

Note that the same callback, `list_cb`, is specified for all three lists.

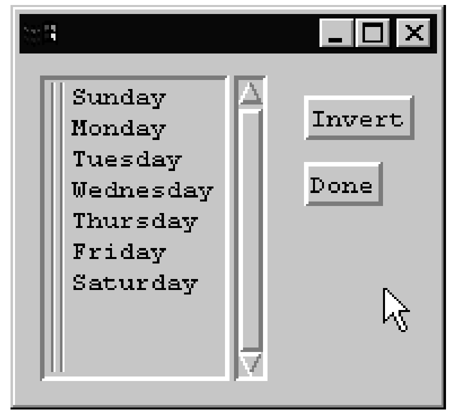
`sel`—A line selection tool

`sel` reads lines on standard input and displays a text list containing the lines. The user then indicates which lines are of interest and then `sel` prints them on standard output.

If the file `days` contains the days of the week, the command

```
sel < days
```

displays this:



Clicking on "Sunday", then "Friday", then `Done` would produce two lines of output. The same clicks, then `Invert`, then `Done`, would produce five lines of output.

This tool might be used to produce an argument list for another command:

```
% rm ‘ls | sel‘
% tar cvf x.tar ‘ls | sel‘
```

(The shell construct `‘x‘` runs the command `x` and substitutes the output in the command line.)

sel—Implementation

In `main`, just above the event-processing loop:

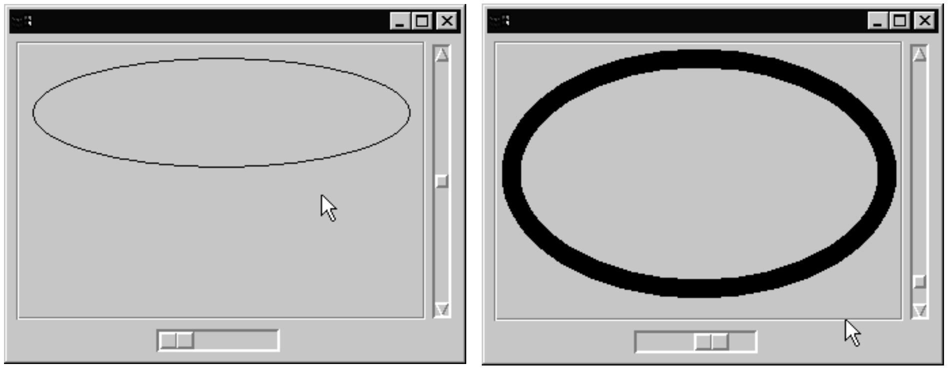
```
every put(items := [], !&input)
VSetItems(vidgets["list1"], items)
```

Callbacks:

```
global selected
procedure list_cb1(vidget, value)
   selected := value
   return
end

#
# To invert the list we first query the state and
# then build a list, 'inverted', that contains
# every position that doesn't appear in the current
# state.
#
# Example: With a five item list, if the state is
# [2,4] then inverted  will be [1,3,5].
#
procedure invert_cb(vidget, value)
    vidget := vidgets["list1"]  # REQUIRED!?!
    selected := VGetState(vidget)

    inverted := [get(selected)] # preserve position

    every i := 1 to *VGetItems(vidget) do
        if not (i = !selected) then put(inverted,i)

    VSetState(vidget, inverted) # calls list_cb1
    return
end

procedure done_cb(vidget, value)
   every write(!selected)
   exit()
end
```


Sliders and scrollbars

Consider a program, `adjust`, that permits adjustment of the "height" and line width of an ellipse via a scrollbar and a slider:



The implementation is simple: Redraw the ellipse whenever the slider or scrollbar is adjusted, using the current state of the two vidgets to control the height and line width.

The slider is configured with a minimum value of 1 and a maximum of 20, directly specifying a line width. The initial value is 1. Filtering is turned off.

The scrollbar uses the default range of 0.0 to 1.0 and an initial value of 0.5. Filtering is turned off.

Sliders and scrollbars, continued

As in the `rpoints` example, a cloned window (`ewin`) that corresponds to the region is established:

```
procedure setup_ewin()
    r := vidgets["region1"]
    ewin := Clone(&window,
        "dx="||r.ux, "dy="||r.uy)
    Clip(ewin, 0, 0, r.uw, r.uh)
end
```

Note that both `ewin` and `vidgets` are declared as globals.

Upon an adjustment we simply call `draw()`, which actually draws the ellipse. The same callback can be used by both the slider and the scrollbar:

```
procedure adjust_cb(vidget, value)
    draw()
    return
end
```

In `main`, we simply create `ewin` and call `draw()` to get the initial ellipse:

```
...
root := vidgets["root"]

setup_ewin()    # Added
draw()          # Added

paused := 1
repeat {
...
```


Sliders and scrollbars, continued

Here is the drawing routine:

```
procedure draw()
    static height, width
    initial {
        width := WAttrib(ewin, "clipw")
        height := WAttrib(ewin, "cliph")
        }

    EraseArea(ewin)

    curheight :=
        VGetState(vidgets["sbar1"]) * (height-20)

    WAttrib(ewin, "linewidth=" ||
        VGetState(vidgets["slider1"]))

    DrawArc(ewin, 10, 10, width-20, curheight)
end
```

`VGetState()` is used to query the positions of both the scrollbar and the slider.

The range of the scrollbar (`sbar1`) is 0.0 to 1.0 and that value is scaled by the height.

The range of the slider is 1 to 20 and that value is used as the line width.

An alternative to the `VGetState()` calls would be to use separate callbacks for the slider and scrollbar. The slider callback would use `WAttrib()` to set the line width. The scrollbar callback would put the value passed to the callback (the second argument) in a global variable that would be accessed in `draw()`.

The 10s and 20s simply provide centering of the ellipse.

Text vidgets

A text vidget consists of a label and a field in which to type text. This application, `text1`, has two text vidgets and, on the line below, a label. To the right of the label is a region with an invisible border.

The concatenation of the text entered in the two text vidgets, and the length, is displayed.

Text vidgets are somewhat limited in functionality. Characters are recognized only when the mouse is over the vidget. A callback is generated only when the user presses return and until the user the presses return, `VGetState()` returns null.

Further, due to a bug in the Windows implementation **<u>the input-sensitive region is not aligned what's drawn on the screen</u>**. The atrocious but currently best workaround for this is to position the window so that the upper left corner of the canvas is in the upper left corner of the display. On Windows NT, this does it:

```
WAttrib("pos=-4,-23")
```


Text vidgets, continued

The callback for each text vidget simply stores the value in a global variable, and calls a routine to update the result:

```
procedure text_input_cb1(vidget, value)
    first_part := value
    show_result()
    return
end
```

`text_input_cb2` is similar, but assigns to `second_part`.

Text vidgets, continued

The common way to produce computed text on a VIB interface is to create an invisible region where the text is to appear and use the coordinates and size of that region to control the output.

Here is a utility routine that is like `WWrites()`, but accepts a region vidget (or its ID) as its first argument and writes the text of the following arguments into the region, truncating appropriately.

```
procedure RWWrites(rvidget, args[])
    r := \vidgets[rvidget] | rvidget
    GotoXY(r.ux, r.uy+WAttrib("ascent"))

    s := ""
    every s ||:= !args
    WWrites(left(s, r.uw/TextWidth("x")))
    return
end
```

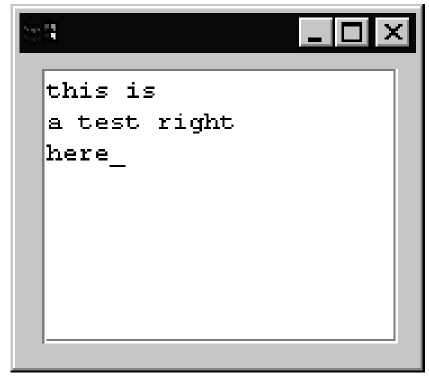
Use `link whmvib` to access the routine.

Here is the routine called by the text vidget callbacks:

```
procedure show_result()
    RWWrites("region1",
        s := first_part || " " || second_part,
        " (", *s, " chars)")
end
```


Example: A text window

Here is an example of using keyboard events from a region to implement a very simple text editing window.




```
procedure region_cb1(vidget, e, x, y)
    static r
    initial r := ""
    case e of {
        default: if type(e) == "string" then {
            case e of {
                "\r": r ||:= "\r\n"
                "\b": if r[-1] == "\n" then
                          r[0-:2] := ""
                      else
                          r[-1] :=  ""
                default:
                    r ||:= e
                }
            EraseArea(point_win)
            GotoRC(point_win,1,1)
            WWrites(point_win,r,"_")
            }
        }
   return
end
```