VIB and Vidgets

Icon has a set of high-level interface objects known as *vidgets* (virtual input gadgets).

The program VIB (visual interface builder) is a WYSIWYG tool for building user interfaces. The command `vib` starts VIB. Here is the initial screen:

![Initial screen of VIB](image)

The icons below the menu bar represent the available vidgets:

- Button
- Radio buttons
- Text list
- Text entry
- Slider
- Scrollbar
- Region
- Label
- Line
VIB, continued

Clicking on vidget's icon causes it to be added to the canvas of the interface. A vidget can be moved with a left-drag and its size can be adjusted by dragging on one of the resize handles.

Here is an interface with several vidgets:

![Interface with vidgets](image)

The overall size of the interface can be adjusted via the target in the lower right hand corner.
Vidget properties

Right-clicking on a vidget brings up a properties dialog for the vidget. Here are the properties for the button:

The *label* is the text displayed on the button.

*ID* is the internal name of the vidget.

*x, y, width, and height* are positioning and sizing information.

*regular, check, etc.* and *outline* specify details of the button's appearance.

*toggle* indicates whether the button stays pressed or rebounds.

*callback* specifies the procedure that is called when the button is pressed.
Vidget properties, continued

Here are the properties for the radio buttons:

A button can be added or removed by clicking the `add` or `del` button in the appropriate position.
Vidget properties, continued

Here are the properties for the slider:

vertical/horizontal specifies the orientation, which can also be changed with the mouse.

top/left and bottom/right indicate the values that correspond to the left- and right-most positions of the thumb. initial is the starting position of the thumb.

filter indicates whether to filter out notifications of position when the slider is being adjusted.
Details on using VIB

If run with no arguments, VIB generates a file named `appl.icn` if no file by that name exists. If `appl.icn` exists, then VIB uses `app2.icn`, and so forth.

If a file is named on the command line, that name is used.

The **File menu operations** `new`, `open`, `save`, `save as`, and `quit` do what their name implies.

`new` and `quit` will warn if changes have been made since the last save, but no such check is made if the windowing system exit is actuated.

The operation **File | refresh** (ALT-R) simply redraws the screen.

The **Edit | copy and delete operations** simply copy or delete the selected vidget. **undelete** undoes the last deletion.

The **Select menu item** simply shows a list of all the widgets that have been placed. Use it to select a vidget that is obscured.
Details on using VIB, continued

The \texttt{Edit} | \texttt{align vert} operation is used to vertically align widgets. To use it:

1. Select a widget.
2. Click on \texttt{Edit} | \texttt{align vert}.
3. Clicking on a widget to cause its Y-coordinate to be set to match the widget selected in the first step.
4. When all vertical adjustments have been made, click on the canvas (i.e., not on a widget) to exit the alignment mode.

The operation of \texttt{Edit} | \texttt{align horz} operation is similar, but for horizontal alignment.

On UNIX systems a different mouse cursor is used when in alignment mode.
Prototype execution with VIB

One of the entries on VIB's File menu is prototype (accessible with ALT-P). This causes generation, compilation and execution of an Icon source file named `vibproto.icn`.

`vibproto.icn` includes a "stub" routine for each vidget's callback. Each stub prints the ID of the vidget and the accompanying data that is passed to the callback. Here's a sample:

```
callback: id=button1, value=1
callback: id=button1, value=1
callback: id=radio_button1, value="A"
callback: id=radio_button1, value="C"
callback: id=slider1, value=0.0
callback: id=slider1, value=1.0
callback: id=region1, value=-1
callback: id=region1, value=-4
callback: id=region1, value="a"
callback: id=region1, value="b"
```
VIB-generated code

Here is the first portion of the file generated for the example at hand:

```
link vsetup

procedure main(args)
    local vidgets, root, paused

    (WOpen ! ui_atts()) | stop("can't open window")
    vidgets := ui()       # set up vidgets
    root := vidgets["root"]

    paused := 1          # flag no work to do
    repeat {
        # handle any events that are available, or
        # wait for events if there is no other work to do
        while (*Pending() > 0) | \paused do {
            ProcessEvent(root, QuitCheck)
        }
        # if <paused> is set null, code can be added here
        # to perform useful work between checks for input
    }
end

Both ui() and ui_attrs() are VIB-maintained procedures.
```
VIB-generated code, continued

The next portion of the file is simply callback routines that do nothing but return:

```
procedure button_cb1(vidget, value)
    return
end

procedure radio_button_cb1(vidget, value)
    return
end

procedure region_cb1(vidget, e, x, y)
    return
end

procedure slider_cb1(vidget, value)
    return
end
```
VIB-generated code, continued

Here is the last portion of the generated file:

```plaintext
#===<<vib:begin>>===    modify using vib; do not remove this marker line
procedure ui_atts()
    return ["size=486,191", "bg=#C0C0C0"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
    [":Sizer::0,0,486,191:",
    ["button1:Button:regular::15,13,32,20:push",button_cb1],
    ["label1:Label:::213,8,54,14:The Region"],
    ["line1:Line:::292,16,382,16:",],
    ["line2:Line:::98,16,188,16:",],
    ["radio_button1:Choice::3:16,43,55,66:",radio_button_cb1,
        ["one","two","three"]],
    ["slider1:Slider:h:1:19,122,51,12:0.0,1.0,0.5",slider_cb1],
    ["region1:Rect:grooved::95,29,289,147:",region_cb1],)
end
#===<<vib:end>>===  end of section maintained by vib
```

The `ui()` routine specifies all the attributes of each widget.

**NOTE:** The main routine and the callbacks are generated only on VIB's initial run for the application. On subsequent runs VIB only manipulates the `ui_atts()` and `ui()` routines.

If you add a vidget in a subsequent run you'll need to edit the file and add a callback routine for it.

BE SURE to exit VIB before manually editing the generated file.
Example: Random points

Consider a VIB-built interface for a program, rpoints, that randomly draws points:

"Clear" is a rebounding button that clears the grid.

"Pause" is a toggle button that pauses drawing.

The radio buttons set the color used for further points.
Drawing in a region

The most complex problem deals with drawing the points in the region.

Each vidget is represented by a record. Every type of vidget except for lines has the fields $ax$, $ay$, $aw$, and $ah$ that describes the rectangle that the vidget covers.

Regions have an additional set of fields, $ux$, $uy$, $uw$, and $uh$ that describe the usable area of the vidget.

The variable $vidgets$ references a table keyed by vidget IDs. (By default it is local but it is sometimes more convenient to make it a global.)

The first modification is in $main$, calling a routine that will cause $point\_win$, a new variable, to reference the usable area of the region:

```haskell
global point\_win    # ADDED
procedure main(args)
  local vidgets, root

  (WOpen ! ui\_atts()) | stop("can't open window")
  vidgets := ui()     # set up vidgets
  root := vidgets["root"]

  point\_win := setup\_point\_win(vidgets) # ADDED
...
```
Drawing in a region, continued

Here is the `setup_point_win` routine:

```cpp
procedure setup_point_win(vidgets)
  local region
  #
  # Get the record representing the region
  #
  r := vidgets["region1"]

  #
  # The subject window is cloned and translation is applied so
  # so that (0,0) in point_win references the upper left corner of
  # the usable area of the region.
  #
  point_win := Clone(&window,"dx="||r.ux, "dy="||r.uy)

  #
  # Clipping is applied so that EraseArea() is limited to the
  # region.
  Clip(point_win, 0, 0, r.uw, r.uh)

  #
  # Use a white background for the region.
  Bg(point_win, "white")
  EraseArea(point_win)

  return point_win
end
```

End result: We can use `point_win` to draw points in the region.
Handling the radio buttons

The next thing is to handle the radio buttons that control the color of the points. We start with a callback routine:

```plaintext
procedure color_cb(vidget, value)
    Fg(point_win, map(value))
    return
end
```

When one of the radio buttons is pressed, `color_cb` is called. `value` will be set to the label of the button that was pressed, i.e., either "Black", "Red", or "White".

The value is mapped to lower case and then `Fg` is called to set the selected color as the foreground color of `point_win`.

We also need another line in `main`:

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

point_win := setup_point_win(vidgets)
VSetState(vidgets["radio_button1"], "Black")  # ADDED
```

The library procedure `VSetState(vidget, value)` sets the state of the specified vidget to the given value.

Calling `VSetState` simulates the effect of the user performing the corresponding operation, so `color_cb` is called.
Handling the Pause button

One element of handling Pause is trivial—a callback routine that sets the global variable \texttt{paused}:

```cpp
procedure pause_cb(vidget, value)
    paused := value
    return
end
```

Because the Pause button is declared as a toggle, \texttt{value} will be 1 when the button is toggled on, and \texttt{null} when toggled off.
Handling the Pause button, continued

The next step is to adjust the event processing loop in main. Here is the original VIB-generated code and comments:

```c
paused := 1  # flag no work to do
repeat {
    # handle any events that are available, or
    # wait for events if there is no other work to do
    while (*Pending() > 0) | \paused do {
        ProcessEvent(root, QuitCheck)
    }
    # if <paused> is set null, code can be added here
    # to perform useful work between checks for input
}
```

The VIB-generated loop accommodates the potential need to interleave other processing with vidget event handling.

Here is a revised version that meets our needs:

```c
paused := &null  # CHANGED
repeat {
    while (*Pending() > 0) | \paused do {
        ProcessEvent(root, QuitCheck)
    }
    draw_points(point_win)  # ADDED
}
```

When the Pause button is toggled on, `pause` is non-null and the application stays in the `while` loop whether there are events pending or not. When Pause is toggled off, `pause` is `&null`, causing execution to drop out of the `while` loop (if no events pending) and call `draw_points()`.

Note that the variable `paused` is generated by VIB but declared as a local. It must be changed to be a global.
Finishing up

Here is the routine \texttt{draw\_points}: 

\begin{verbatim}
procedure draw_points(W)
    static width, height
    initial {
        width := WAttrib(W, "width")
        height := WAttrib(W, "height")
    }

    every 1 to 100 do
        DrawPoint(W, ?width, ?height)
    end
\end{verbatim}

Finally, here is a callback for the Clear button: 

\begin{verbatim}
procedure clear_cb(vidget, value)
    EraseArea(point_win)
    return
end
\end{verbatim}

Because the Clear button is a rebounding button, \texttt{value} is always 1.
Pausing with a click in the points

Problem: Modify the program so that a left click in the points has the same effect as toggling the Pause button on. A right click in the points toggles the Pause button off.

Here is the callback for the region:

```plaintext
procedure region_cb1(vgrid, e, x, y)
  return
end
```

Note that the callback for a region is passed the event and the coordinates of the event.
rpoints: Complete source

For reference, here is the complete source for rpoints.

```lisp
link vsetup
global point_win
global paused # CHANGED
global vidgets # CHANGED
procedure main(args)
  local root # CHANGED
  (WOpen ! ui_atts()) | stop("can't open window")
  vidgets := ui() # set up vidgets
  root := vidgets["root"]

  point_win := setup_point_win(vidgets)
  VSetState(vidgets["radio_button1"], "Black")

  paused := &null # CHANGED!
  repeat {
    # handle any events that are available, or
    # wait for events if there is no other work to do
    while (*Pending() > 0) | \paused do {
      ProcessEvent(root, QuitCheck)
    }
    # if <paused> is set null, code can be added here
    # to perform useful work between checks for input
    draw_points(point_win)
  }
end
procedure setup_point_win(vidgets)
  local region
  r := vidgets["region1"]
  point_win := Clone(&window,
    "dx="||r.ux, "dy="||r.uy)
  Clip(point_win, 0, 0, r.uw, r.uh)
  Bg(point_win, "white")
  EraseArea(point_win)
  return point_win
end
```
rpoints: Complete source, continued

procedure draw_points(W)
    static width, height
    initial {
        width := WAttrib(W, "width")
        height := WAttrib(W, "height")
    }

    every 1 to 100 do
        DrawPoint(W, ?width, ?height)
    end

procedure clear_cb(vidget, value)
    EraseArea(point_win)
    return
end

procedure pause_cb(vidget, value)
    paused := value
    return
end

procedure color_cb(vidget, value)
    Fg(point_win, value)
    return
end

procedure region_cb1(vidget, e, x, y)
    case e of {
        &lpress: VsetState(vidgets["button2"], 1)
        &rpress: VsetState(vidgets["button2"], &null)
    }
    return
end
rpoints: Complete source, continued

#===<<vib:begin>>=== modify using vib; do not remove this marker line

procedure ui_atts()
    return ["size=378,198", "bg=#C0C0C0"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
    [":Sizer::0,0,378,198:",]
    ["button1:Button:regular::285,9,84,20:Clear",clear_cb],
    ["button2:Button:regular:1:287,43,84,20:Pause",pause_cb],
    ["radio_button1:Choice::3:288,74,57,66:",color_cb,
        ["Black","Red","White"]],
    ["region1:Rect:grooved::8,6,262,182:",region_cb1],
)
end

#===<<vib:end>>=== end of section maintained by vib