The Icon Analyst

In-Depth Coverage of the Icon Programming Language

June 1997
Number 42

In this issue …

Multiple VIB Interfaces

Editors' Note: This article is based in part on material provided by Clint Jeffery.

A single visual interface window is adequate and appropriate for most applications. There are situations, however, when more than one interface is needed. Typical examples are multi-user games and painting and drawing applications.

Before designing an application with more than one interface window, consider the problems: managing multiple windows adds programming complexities, and in single-user situations an application with more than one window requires the user to change his or her focus of attention. In addition, applications with multiple windows require more screen space than single-window applications.

VIB Considerations

VIB can handle only one interface section in a file. There are ways of fooling VIB by editing the lines it places at the beginning and end of its interface code, but these are clumsy. For multiple interfaces, then, it's usually best to put each one in a separate file. The files then can be linked in the application, as in

```
link control    # control window
link draw       # drawing window
```

where control.icn and draw.icn are two visual interfaces and have been compiled into ucode files, as in

```
icont –c control.icn draw.icn
```

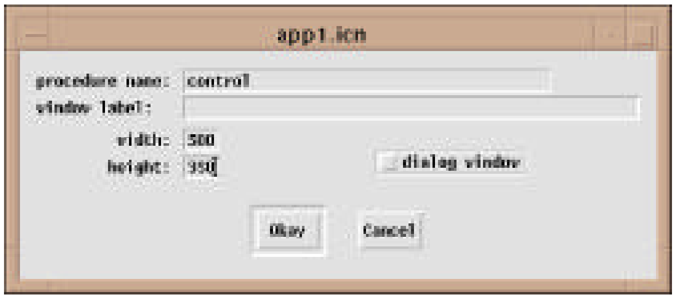
(When more than one file is compiled with the –c option, separate ucode files are created for each one.)

When VIB creates a file, as opposed to modifying an existing one, it provides a main procedure that is useful for testing. In the program organization we're using here, such main procedures should be deleted and one provided in the program that links the interface code. (VIB does not add a main procedure when editing an existing file, so this only needs to be done once.)

The code for a VIB interface contains two procedures that are named ui_atts and ui by default. The procedure ui_atts() returns the attributes used to open the interface window. In most applications, it is not needed, but it can be used to open the window with added or changed attributes. The procedure ui() opens the interface window if &windows is null, draws its vidgets, and initializes the interface.

To avoid conflicting declarations for these procedures in multiple interfaces, their names need to be changed. This is done easily in VIB by specifying a procedure name in the canvas dialog as shown in Figure 1 at the top of the next page. The name specified — control in this case — is used in place of ui for the two procedures in the interface code. In the example above, they are named control_atts() and control().

Each interface has its own vidgets. The same vidget ID can be used in more than one interface, but care should be taken not to use the same callback name in more than one interface unless a single procedure handles callbacks from more than one window.


app1.icn

procedure name: control

window label:

width: 500

height: 350

dialog window

Okay Cancel

Figure 1. A VIB Canvas Dialog

If the names are changed to draw_atts() and draw() in draw.icn, the application might begin as follows:

```
control_vidgets := control()
              …
draw_vidgets := draw()
```

Note that each interface produces its own table of vidgets.

There's a "gotcha" here: As mentioned earlier, a "ui" procedure only opens a window if &window is null. If &window is null when control() is called, and nothing else is done, draw() does not open a window, but instead overwrites what control() drew in the window it opened. This problem is easily fixed:

```
control_vidgets := control()
&window := &null
draw_vidgets := draw()
```

If there is need to refer to the windows later in the program, they can be assigned to variables as follows:

```
control_vidgets := control()
control_win := &window
&window := &null
draw_vidgets := draw()
draw_win := &window
```

An alternative to setting &windows to the null value between calls of the "ui" procedures is to open the windows using the "ui_atts" procedures:

```
&window := WOpen ! control_atts()
control_vidgets := control()
&window := WOpen ! draw_atts()
draw_vidgets := draw()
```

The window to use also can be specified as the argument to a "ui" procedure, as in

```
control_win := WOpen ! control_atts()
control_vidgets := control(control_win)
draw_win := WOpen ! draw_atts()
draw_vidgets := draw(draw_win)
```

Controlling Multiple Interfaces

As mentioned earlier, each interface has its own vidgets; in each, the ID of the root vidget that encloses and manages all others in the interface is "root". These roots can be obtained as needed or assigned to variables, as in

```
control_root := control_vidgets["root"]
draw_root := draw_vidgets["root"]
```

The most difficult part remains: managing events in more than one window. How this is done depends on the functionality of the application.

The simplest case is a purely event-driven application in which actions are taken only in response to user events and events in all interface windows have equal priority and need to be handled as they occur.

In this case, it is not sufficient to process the windows in order, waiting, for example, for an event in the first window before going on to the second. If this is done, events may accumulate in other windows and not be processed.

The function Active() can be used to deal with this problem. Active() returns a window in which an event is pending, blocking and waiting for an event if none is pending. Every time Active() is called, it starts with a new window in round-robin fashion, to assure that all windows can be serviced.

The event loop for an event-driven application of the kind described above might look like this:

```
repeat {
   root := case Active() of {
      control_win:  control_root
      draw_win:     draw_root
      }
   ProcessEvent(root, …)
   }
```

where the ellipses indicate other possible arguments for ProcessEvent().

An alternative to assigning the interface windows to variables is to make use of the fact that a root vidget has a win field that is the window with which it is associated. Therefore, the event loop above can be rewritten as follows:

```
repeat {
   root := case Active() of {
      control_root.win:   control_root
      draw_root.win:      draw_root
      }
   ProcessEvent(root, …)
   }
```

The code can be further collapsed by putting the case expression as the first argument of ProcessEvent():

```
repeat {
  ProcessEvent(
     case Active() of {
        control_root.win:    control_root
        draw_root.win:       draw_root
        },
     …        # other arguments for ProcessEvent()
     )
   }
```

In some applications, different interface windows may have different priorities. For example, a drawing application might be designed so that there is a shift in focus between the control window and the drawing window. Furthermore, when the drawing window is the focus, all events in it might be processed, ignoring events in the control window until a specific event in the drawing window changes the focus to the control window, and vice versa. The code might look like this:

```
   root := control_root              # initial interface
   while ProcessEvent(root, …)
                     …

procedure go_draw()                  # callback in control

   root := draw_root

   return

end

procedure go_control()               # callback in draw

   root := control_root

   return

end
```

where go_draw() is in control.icn and go_control() is in draw.icn.

One problem with this is that if events occur in the control window while the draw window is the focus of attention, these events are not processed until the focus is changed — and then they all are processed.

One way to handle this problem is to discard events that occur in windows other than the focus window. This can be done by emptying the event queue of the window that is to become the focus before changing the focus. The callbacks given earlier can be modified to do this:

```
procedure go_draw()              # callback in control

   while get(Pending(draw_win))

   root := draw_root

   return

end

procedure go_control()           # callback in draw

   while get(Pending(control_win))

   root := control_root

   return

end
```

Handling multiple interfaces in an application like the kaleidoscope [1-2] that is not entirely event driven poses other problems.

In this kind of an application, processing goes on even if there are no user events, but user events must be processed when they occur.

For a single interface, the event-processing loop typically looks something like this:

```
repeat {
   while   Pending() > 0 do
      while ProcessEvent(root, …)
      # do something before checking for next event
   }
```

It's important that what's done before checking for the next event be brief; otherwise the user may become annoyed at the unresponsiveness of the interface, perhaps repeat actions that "didn't take", or even assume the application is hung.

If we introduce multiple interfaces, this event loop needs to be recast. For two interfaces, the loop might look like this:

```
repeat {
   while   Pending(win1 | win2) > 0 do
      ProcessEvent(
      case Active() of {
         win1:     root1
         win2:     root2
         },
      …
      )
      # do something before checking for next event
   }
```

Note that Active() is called only if there is an event pending in one of the windows; it therefore does not block.

References

1. “The Kaleidoscope”, *Icon Analyst* 38, pp. 8-13.

2. “The Kaleidoscope”, *Icon Analyst* 39, pp. 5-10.

Debugging: Tracing

Tracing is Icon’s most powerful debugging tool. Tracing produces messages for both procedure and co-expression activity.

Procedure Tracing

For procedures trace messages occur for

invocation (call)
return
failure
suspension
resumption

Consider this recognizer for context-sensitive strings of the form $a^n b^n c^n$ [1]:

```
procedure main()

   while writes(line := read()) do
      if line ? {
         abc("") & pos(0)
         }
      then write(" accepted") else write(" rejected")

end

procedure abc(s)

   suspend =s | (="a" || abc("b" || s) || ="c")

end
```

For the input abcc (which is not recognized), the trace output is:

```
                    :       main()
abc.icn             :   5   | abc("")
abc.icn             :  13   | abc suspended ""
abc.icn             :   5   | abc resumed
abc.icn             :  13   | | abc("b")
abc.icn             :  13   | | abc suspended "b"
abc.icn             :  13   | abc suspended "abc"
abc.icn             :   5   | abc resumed
abc.icn             :  13   | | abc resumed
abc.icn             :  15   | | abc failed
abc.icn             :  15   | abc failed
abc.icn             :   9   main failed
```

Notice that program execution begins with a call to main(). If program execution terminates by a return, failure, or suspension from main(), there is a trace message for this, as shown in the example above.

Except for the call to main(), which does not occur in the program itself, each line of tracing shows the name of the source file in which procedure activity occurs (abc.icn in the example above). If the file is composed from more than one file, that is reflected in the name field. The field for the file name is 13 characters long. If the file name is longer than that, the initial part is discarded (it probably should be the trailing part).

Following a separating colon after the file name, the line number in the file where the activity occurred is given, followed by vertical bars that indicate the level of procedure call (the value of &level).

The rest of the line indicates the type of procedure activity with an associated value if the activity has one. Values are shown as they are for error traceback [2].

If you recall the article on event monitoring for functions [3], you’ll notice there is no trace message for removal of a suspended procedure on exit from a bounded expression. There is such an event for procedures, and removal is important, because otherwise suspended procedures would accumulate, consuming more and more memory. The removal of a suspended procedure is an implementation matter, however, not a language feature. In interpreting trace output, it is important to remember this, because a suspended procedure may never be resumed. Here is an example using a recursive generator given in Reference 4:

```
procedure main()

   while writes(line := read()) do
      if line  == star('abc') \ 10
      then write(" yes") else write(" no")

end
```

```
procedure star(chars)
   suspend  "" | (star(chars) || !chars)
end
```

Since star() is a infinite recursive generator, some limit on its results is needed; otherwise, in the case of input that is not included in its sequence, stack overflow eventually occurs.

Here is the trace output for the input aa, which is found:

```
             :      main()
findstar.icn :  3  |  star('abc')
findstar.icn :  9  |  star suspended ""
findstar.icn :  3  |  star resumed
findstar.icn :  9  |  | star('abc')
findstar.icn :  9  |  | star suspended ""
findstar.icn :  9  |  star suspended "a"
findstar.icn :  3  |  star resumed
findstar.icn :  9  |  star suspended "b"
findstar.icn :  3  |  star resumed
findstar.icn :  9  |  star suspended "c"
findstar.icn :  3  |  star resumed
findstar.icn :  9  |  | star resumed
findstar.icn :  9  |  |  | star('abc')
findstar.icn :  9  |  |  | star suspended ""
findstar.icn :  9  |  | star suspended "a"
findstar.icn :  9  |  star suspended "aa"
findstar.icn :  5   main failed
```

Note that the last suspension of star() is not resumed.

Tracing occurs if the value of &trace is nonzero. (The initial default value of &trace is 0, except as noted in the following paragraph.) The value of &trace is decremented for each trace message, so if &trace is set to a positive value, it turns off automatically, provided procedure activity continues long enough. Assigning a negative value to &trace, as in

```
&trace := –1
```

results in unlimited tracing. The value of &trace can, of course, be changed at any time during program execution.

The initial value of &trace also can be set in a command-line environment by compiling a program with the with the –t option, as in

```
icont –t example
```

which sets the initial value of &trace to –1. The initial value of &trace also can be set by using the environment variable TRACE to a nonzero value, as in

```
setenv TRACE  –1
```

Values other than –1 can be used.

Since these methods only affect the initial value of &trace, setting it in a program can be used to override the effect of the initial value (although if the initial value of &trace is nonzero, there still is a trace message for the call to main() that starts program execution).

A problem with the –t option is that tracing occurs when the program is run, but there is no evidence of tracing in the source program. The environment variable TRACE overrides –t, so

```
setenv TRACE  0
```

turns off tracing in a program compiled with –t, unless &trace is set in the program itself.

A problem with setting TRACE to a nonzero value is that it affects all Icon programs — even programs you may not know were written in Icon.

Procedure tracing suffers from the same problems that termination dumps do, except that procedure tracing may be arbitrarily long:

• Output always is sent to standard error output.

• Output may be voluminous and contain messages for procedures that are not in the program proper, especially in programs that use graphics and have VIB interfaces.

Tracing also is nonselective; all procedures produce messages. These problems can be mitigated to some extent by facilities in the Icon program library, which we will describe in a later article.

In some situations, it may be helpful to limit the number of trace messages by setting the value of &trace to a small positive value at a selected location, as in

```
&trace := 10
# trace this suspicious case
        …
```

Co-Expression Tracing

For co-expresions, trace messages occur for

activation

```
    return
    failure
```

Consider this program, which prepends a label to lines of input that begin with a blank:

```
procedure main()

   digit := create seq()

   while line := read() do {
      if line[1] == " " then writes("L", right(@digit, 3, "0"), ":")
      write(line)
      }

end
```

Here is typical trace output:

```
               :      main()
label.icn      :   6  | main; co-expression_1 : &null @ co-expression_2
label.icn      :   3  | main; co-expression_2 returned 1 to co-expression_1
label.icn      :   6  | main; co-expression_1 : &null @ co-expression_2
label.icn      :   3  | main; co-expression_2 returned 2 to co-expression_1
                      …
```

The level shown is for procedures; there is no concept of level for co-expression activity. Note that the unary @C operation is just an abbreviation for &null @ C. Note also that trace messages are produced for both procedure activity and co-expression activity; there is no way to select only one kind. The lines are rather long because of two occurrences of co–expression in each of them; we've made the type size smaller to make it possible to compose this page.

In the example above, the use of co-expressions is simple. This need not be the case, as illustrated by this contrived example in which there is no predictable order to co-expression activation:

```
global co_list

procedure main()

   co_list := []

   every 1 to 4 do
      put(co_list, create |(?&letters @ ?co_list))

   @?co_list

end
```

Typical trace output from this program is

```
               :      main()
chaos.icn      :  10  | main; co-expression_1 : &null @ co-expression_2
chaos.icn      :   8  | main; co-expression_2 : "c" @ co-expression_3
chaos.icn      :   8  | main; co-expression_3 : "q" @ co-expression_4
chaos.icn      :   8  | main; co-expression_4 returned "q" to co-expression_3
chaos.icn      :   8  | main; co-expression_3 : "m" @ co-expression_2
chaos.icn      :   8  | main; co-expression_2 returned "m" to co-expression_3
                      …
```

Needless to say, it may be difficult to interpret tracing of unstructured co-expression activity. In our experience co-expression tracing is generally less useful than procedure tracing for debugging

References

1. *The Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, 1996, pp. 223-224.

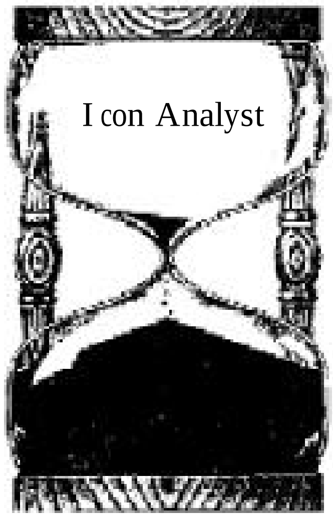
2. "Debugging: Built-In Facilities", Icon Analyst 41, pp. 4-7.

3. "Dynamic Analysis of Icon Programs", Icon Analyst 28, pp. 9-12.

4. *The Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Peer-to-Peer Communications, 1996, p. 210.

Next Time

In the next article on debugging, we'll start describing the facilities in the Icon program library that can be used in conjunction with the built-in facilities to make their output easier to use.



Renew Your Subscription Now

From the Library: Anatomy of a Graphics Procedure

In this article we'll examine in detail a graphics procedure from the Icon program library [1]. We'll look closely at the code and discuss many of the techniques. We'll also consider some of the various design decisions. At the end, we'll give a complete listing of the procedure.

The procedure of interest implements *dragging*, which is found in many graphical interfaces. Dragging is a process in which the user manipulates the mouse to move an object in a window from one place to another. An object, in this sense, is generally something visually distinct that can be moved as a rigid unit while the rest of the display remains static. An icon on a desktop is an example.

Dragging begins by positioning the mouse pointer over an object and pressing a mouse button. With the button down, the mouse is moved, and the object moves with it, so that the same point on the object stays under the mouse pointer. Finally, the mouse button is released to "let go" of the object and leave it in its new position. In the before-and-after figures at the top of the next column, the center square is dragged out of its column to a new position.

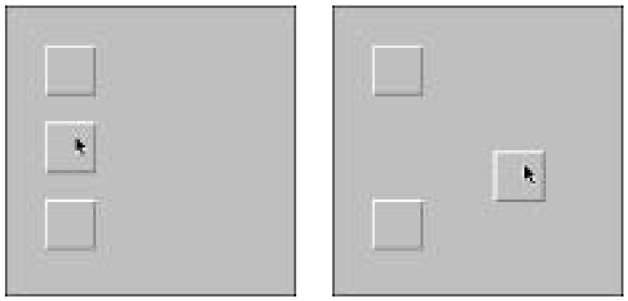


Figure 1. Dragging a Rectangle

Dragging is implemented in the Icon program library by the procedure Drag(). It was originally written over two years ago; this discussion is necessarily a reconstruction of our thoughts at that time, omitting dead-end approaches and other problems that we've now forgotten.

Motivation

As part of a VIB face-lift [2], we wanted better realism during the rearrangement of interface objects. Most of these objects are rectangular. Before, when such an object was moved, an outline followed the mouse until it was placed. We wanted instead to move the entire object, not just its outline, to more faithfully represent the appearance of the interface during movement.

Because dragging is a standard interactive paradigm, it seemed wise to produce a general-purpose procedure rather than just something specific to VIB. This predisposition towards writing sharable code has led to several library proce-

dures.

The Drag() procedure was written and debugged along with a special test driver, then installed in the library and used by VIB.

The Design

Coming up with a design took a bit of thought. Unlike most graphics procedures, dragging involves both input and output. What we settled on is given by the comments at the front:

> Drag(x, y, w, h) lets the user move a rectangular area using the mouse. Called when a mouse button is pressed, Drag() handles all subsequent events until a mouse button is released. As the mouse moves, the rectangular area originally at (x, y, w, h) follows it across the screen; vacated pixels at the original location are filled with the background color. The rectangle cannot be dragged off-screen or outside the clipping region. When the mouse button is released, Drag() sets &x and &y to the upper-left corner of the new location and returns.

Let's review that one sentence at a time.

Drag(x, y, w, h) lets the user move a rectangular area using the mouse. Drag() is associated with user interaction, and it deals with a rectangular area specified in the conventional manner. Rectangular objects are a common case, and we weren't prepared to attack the much harder problem of dragging arbitrary shapes.

Called when a mouse button is pressed, Drag() handles all subsequent events until a mouse button is released. The assumption is that the user initiates dragging by pressing a mouse button while the pointer is atop an object to be dragged. The program that calls Drag() is expected to identify an event as a dragging request and determine the location and extent of the corresponding object before making the call. The Drag() procedure, in turn, reads all subsequent events up to and including the first release of any mouse button, which signals the end of dragging.

As the mouse moves, the rectangular area originally at (x, y, w, h) follows it across the screen. While Drag() is active, it moves the image of the object being dragged — that rectangular area — to correspond to the mouse motion.

Vacated pixels at the original location are filled with the background color. When the object moves away from its original position, what should replace it in the area that has been "exposed"? We decided to simply paint those pixels with the window background color, as if the object had been sitting in front of a plain canvas. That is not an ideal solution, because it does not handle the exposure of overlapped objects, but we decided it was good enough. (VIB sets a darker-than-normal background color before calling Drag(), leaving a visible hole, and then fills it in when Drag() returns.)

The rectangle cannot be dragged off-screen or outside the clipping region. These restrictions simplify the coding. As long as the object remains within the intersection of the clipping region and the window boundaries, it can be moved around using CopyArea(). If it were allowed to move outside this region, part of it would disappear; and without additional complication, we'd have no way to get it back. The fact that Drag() notices the clipping region also could be considered a feature, in that it gives the caller some additional control.

When the mouse button is released, Drag() sets &x and &y to the upper-left corner of the new location and returns. How does Drag() tell its caller where the object ended up? It needs to return two values, an x value and a y value. Rather than encoding both values somehow in the procedure return value, we decided to set the values of the keywords &x and &y. That makes a certain amount of sense, because those keywords are set by input events, and Drag() is just a fancy event handler. The returned values of &x and &y specify the object location, not the final mouse position.

The Code Prologue

We will now go through the body of Drag() from beginning to end, looking (at least briefly) at every line of code. It's a large procedure, but we'll break it into small chunks and give a short explanation before each code fragment. The complete procedure is listed in Figure 2 on pages 10 and 11.

The first half of the code sets things up for an event loop. The code begins, of course, with the procedure declaration. Drag() actually declares five

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

parameters — a win parameter in addition to the rectangle specification. Like most of the library's graphics procedures, an optional leading window specification is allowed, and this additional parameter is not mentioned in the individual procedure's documentation.

```
procedure Drag(win, x, y, w, h)
   (local declarations)
```

An idiomatic approach is used to allow omission of the window argument. If a window is supplied, nothing is done and the code proceeds onward. If a window is not supplied, a recursive call is made, shifting the values of the win, x, y, w parameters into the x, y, w, h positions. The recursive call specifies &window for the window value, if it is not null; otherwise runtime error 140 is reported ("window expected") and the recursive call never occurs.

```
   if type(win) ~== "window" then
      return Drag((\&window | runerr(140)), win, x, y, w)
```

No provision is made for defaulting x, y, w, or h; all must be specified.

Icon allows negative values in width and height specifications. The following code converts negative w and h values to positive equivalents, with corresponding adjustments to x and y.

```
if w < 0 then
   x –:= (w := –w)
if h < 0 then
   y –:= (h := –h)
```

A good library procedure should not assume that the window's attributes have default values. If the caller has changed the dx or dy attributes, the window's coordinate system has changed, affecting subsequent calculations.

```
dx := WAttrib(win, "dx")
dy := WAttrib(win, "dy")
```

Initial limits of motion are calculated from the window size and dx/dy values. The variables x0 and x1 hold the horizontal limits of movement for the object's upper-left corner, and similarly y0 and y1 hold the vertical limits.

```
x0 := –dx
y0 := –dy
x1 := WAttrib(win, "width") – dx – w
y1 := WAttrib(win, "height") – dy – h
```

If a clipping region has been set, the motion limits are further adjusted. New values are assigned only if they make the region smaller.

```
x0 <:= \WAttrib(win, "clipx")
y0 <:= \WAttrib(win, "clipy")
x1 >:= \WAttrib(win, "clipx") +
   \WAttrib(win, "clipw") – w
y1 >:= \WAttrib(win, "clipy") +
   \WAttrib(win, "cliph") – h
```

A "scratch canvas" is obtained using another library procedure not discussed here. This is just a hidden window used for refreshing areas exposed by movement of the dragged object. CopyArea() is called to initialize the hidden window with a copy of the visible window.

```
behind := ScratchCanvas(win)
CopyArea(win, behind, –dx, –dy)
```

As discussed earlier, moving the object from its original position exposes the background color. This is actually accomplished by setting the corresponding portion of the scratch canvas to the background color.

```
Bg(behind, Bg(win))
EraseArea(behind, x + dx, y + dy, w, h)
```

In order to keep the same point of the object under the mouse pointer as it is dragged, the offset from that point to the object corner is recorded; &x and &y contain the mouse position of the event that initiated dragging.

```
xoff := x – &x
yoff := y – &y
```

That completes the preparations; the remaining code awaits and reacts to input events.

The Event Loop

Event processing is handled by a loop that executes once for each event. The loop begins by

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

getting the next event from Event() and checking it. Any of the three kinds of mouse release events causes the loop to terminate. For any other kind of event, the loop body is entered.

```
  until Event(win) ===
     (&lrelease | &mrelease |
        &rrelease) do {
```

Any other event is most likely a mouse movement event; but the actual event doesn't matter, and isn't even recorded, as long as it is not a mouse button release. What counts is the location at which the event occurred.

The new object position is calculated from the pointer location and the recorded offset. The position is constrained to be within the limits given by x0, x1, y0, and y1, and the object is drawn at its new location by copying its image from the old location.

```
     # move the rectangle
     xnew := &x + xoff
     ynew := &y + yoff
     xnew <:= x0
     ynew <:= y0
     xnew >:= x1
     ynew >:= y1
     CopyArea(win, x, y, w, h,
        xnew, ynew)
```

The area exposed by the latest movement is now redrawn by copying from the scratch canvas. This is easy if the new and old locations do not overlap.

```
     xshift := xnew – x
     yshift := ynew – y

     if abs(xshift) >= w |
        abs(yshift) >= h then {
           CopyArea(behind, win,
              x + dx, y + dy, w, h, x, y)
        }
```

Several different situations must be considered when there

```
procedure Drag(win, x, y, w, h)
   local dx, dy, x0, y0, x1, y1
   local behind, xoff, yoff, xnew, ynew, xshift, yshift

   if type(win) ~== "window" then
      return Drag((\&window | runerr(140)), win, x, y, w)

   if w < 0 then
      x –:= (w := –w)
   if h < 0 then
      y –:= (h := –h)

   dx := WAttrib(win, "dx")
   dy := WAttrib(win, "dy")

   x0 := –dx                                   # set limits due to window size
   y0 := –dy
   x1 := WAttrib(win, "width") – dx – w
   y1 := WAttrib(win, "height") – dy – h

   x0 <:= \WAttrib(win, "clipx")               # adjust limits for clipping
   y0 <:= \WAttrib(win, "clipy")
   x1 >:= \WAttrib(win, "clipx") + \WAttrib(win, "clipw") – w
   y1 >:= \WAttrib(win, "clipy") + \WAttrib(win, "cliph") – h

   behind := ScratchCanvas(win)
   CopyArea(win, behind, –dx, –dy)
   Bg(behind, Bg(win))
   EraseArea(behind, x + dx, y + dy, w, h)

   xoff := x – &x
   yoff := y – &y

   until Event(win) === (&lrelease | &mrelease | &rrelease) do {

      # move the rectangle
      xnew := &x + xoff
      ynew := &y + yoff
      xnew <:= x0
      ynew <:= y0
      xnew >:= x1
      ynew >:= y1
      CopyArea(win, x, y, w, h, xnew, ynew)

      # repaint the area exposed by its movement
      xshift := xnew – x
      yshift := ynew – y

      if abs(xshift) >= w | abs(yshift) >= h then {

         # completely disjoint from new location
         CopyArea(behind, win, x + dx, y + dy, w, h, x, y)
         }

      else {

         # new area overlaps old
```

Figure 2. The Procedure Drag()

is overlap. Vertical and horizontal movement are handled independently, although this may do a slight amount of extra copying.

```
else {
 if xshift > 0 then
  CopyArea(behind, win,
    x + dx, y + dy, xshift, h, x, y)
else if xshift < 0 then
  CopyArea(behind, win,
     x + dx + w + xshift, y + dy,
     –xshift, h, x + w + xshift, y)
if yshift > 0 then
  CopyArea(behind, win,
     x + dx, y + dy, w, yshift,
     x, y)
else if yshift < 0 then
  CopyArea(behind, win,
     x + dx, y + dy + h + yshift,
     w, –yshift, x, y + h + yshift)
}
```

Finally, at the bottom of the loop, x and y are updated to reflect the new object location, and the loop repeats.

```
x := xnew
y := ynew
}
```

Finishing Up

Looping ends when a mouse button is released. Any mouse movement was seen earlier in the form of mouse-drag events, so the x and y values of the release event can be ignored.

An EraseArea() call ensures that the scratch canvas does not tie up any unneeded color map entries. The keywords &x and &y are set in order to pass back the final object location. As is conventional for graphics procedures, the window value is returned.

```
EraseArea(behind)
&x := x
&y := y
return win
```

Conclusion

We've looked over a graphics procedure from the Icon program library and examined it in detail. Many of the techniques and considerations noted here apply also to other library procedures.

If written just for VIB, this procedure could have been smaller and simpler. VIB does not use a defaulted window argument, negative size specifications, or a translated coordinate system.

On the other hand, the extra effort to address these concerns leads to fewer surprises for its potential callers and makes Drag() more suitable for inclusion in the library.

```
      if xshift > 0 then
        CopyArea(behind, win, x + dx, y + dy, xshift, h, x, y)
      else if xshift < 0 then
        CopyArea(behind, win,
          x + dx + w + xshift, y + dy, –xshift, h, x + w + xshift, y)
      if yshift > 0 then
        CopyArea(behind, win, x + dx, y + dy, w, yshift, x, y)
      else if yshift < 0 then
        CopyArea(behind, win,
          x + dx, y + dy + h + yshift, w, –yshift, x, y + h + yshift)
      }

    x := xnew
    y := ynew
    }

  EraseArea(behind)

  &x := x
  &y := y

  return win

end
```

Figure 2 (continued). The Procedure Drag()

References

1. *The Icon Program Library; Version 9.3*, Ralph E. Griswold and Gregg M. Townsend, Icon Project Document IPD279, Tucson, Arizona, 1996.

2. *VIB: A Visual Interface Builder for Icon; Version 3*, Gregg M. Townsend and Mary Cameron, Icon Project Document IPD265, Tucson, Arizona, 1996.

Downloading Icon Material

Most implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

Subscription Renewal

For many of you, this is the last issue in your present subscription to the Analyst. If so, you'll find a renewal form in the center of this issue. Renew now so that you won't miss an issue.

Your prompt renewal helps us by reducing the number of follow-up notices we have to send. Knowing where we stand on subscriptions also lets us plan our budget for the next fiscal year.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA ®
TUCSON ARIZONA

and

Bright Forest Publishers

Tucson Arizona

Analyst on Diskettes?

We only received two responses to our proposal to make the Analyst available on diskettes in Adobe Portable Document Format.

One response was enthusiastically positive. The other posed a number of questions and raised several troublesome issues.

Although we believe the issues can be resolved, with the lack of response from other readers, we're not going to undertake the project — at least in the near future.

If you have thoughts on the subject that you've not expressed, please let us know what they are.

What's Coming up

In the next issue of the Analyst, we'll continue the series on debugging with a description of tools in the Icon program library that can make the built-in facilities more useful.

We still haven't completed the article on factors of versum numbers; maybe for next time.

We plan to get back to program monitoring in the next Analyst, with an emphasis on presenting results visually.