Interaction basics

Certain actions by the user of a graphical Icon program cause events to be produced.

Events fall into three categories: keystrokes, mouse actions, and window resizing.

The Event() function returns the next event from the event queue. If the queue is empty, Event() waits.

Mouse events are represented by keywords such as &lpress and &rrelease.

A simple example:

```c
procedure main() # ev1
    WOpen("size=300,400")
    repeat {
        case Event() of {
            &lpress: WWrite("left button down")
            &lrelease: WWrite("left button up")
            &rpress: break
        }
    }
end
```
Interaction basics, continued

Each event is actually represented by three values: an event code, and x and y coordinates.

Event() returns the code for the next event and as a side effect sets &x and &y. For mouse events the code is a small negative integer, such as −1 for &lpress.

Here is a program that identifies the quadrant in which the left button was clicked:

```plaintext
procedure main() # ev2
    WOpen("size=300,300")
    DrawSegment(150,0,150,300,0,150,300,150)
    repeat {
        case Event() of {
            &lpress: {
                if &y < WAttrib("height")/2 then
                    WWrites("Upper ")
                else
                    WWrites("Lower ")
                if &x < WAttrib("width")/2 then
                    WWrite("left")
                else
                    WWrite("right")
            }
            &rpress: break
        }
    }
end

Recall that DrawSegment draws non-contiguous lines.
```
Interaction basics, continued

Here is a very simple drawing program from the text, page 185:

```plaintext
procedure main() # ev3
    WOpen("size=400,300")
repeat {
    case Event() of {
        &lpress: {
            DrawPoint(&x, &y)
            x := &x
            y := &y
        }
        &ldrag: {
            DrawLine(x, y, &x, &y)
            x := &x
            y := &y
        }
        &rpress|&rdrag:
            EraseArea(&x - 2, &y - 2, 5, 5)
    }
end
```

Problem: Describe what would be necessary to save and load drawings.
Interaction—keystroke events

Keystrokes produce events. For keys such as A, $, 4, ?, and =, the value produced by \texttt{Event()} is a string that corresponds to the key. For other keys, such as the function keys and cursor keys, \texttt{Event()} produces an integer.

```pascal
procedure main() # key1
    WOpen("size=300,400")
    repeat {
        case e := Event() of {
            "q"|"Q": break
            default: WWrite(image(e))
        }
    }
end
```

The library file \texttt{keysyms.icn} has \$\texttt{defines} for various non-textual keys. Examples:

- \$\texttt{define Key_Home} \hspace{1cm} 36
- \$\texttt{define Key_Insert} \hspace{1cm} 45
- \$\texttt{define Key_F1} \hspace{1cm} 112

Use \texttt{$include "keysyms.icn"} (not link).

Keystrokes and mouse actions can be intermixed:

```pascal
    case Event() of {
        &lpress: ... 
        !"Qq"|&rpress: ...
    }
```
Interaction—keystrokes, continued

Just like with mouse events, \&x and \&y are set when a keystroke event is fetched with Event().

The keywords \&control, \&shift, and \&meta can be used to test whether the control, shift, and/or meta (ALT) keys were pressed in conjunction with generation of the event.

The keyword \&interval is set to the number of milliseconds that elapsed between this event and the last event.

This program shows information about events:

```pascal
procedure main() # key3 (based on p.187 of text)
  WOpen("size=300,400")
  repeat {
    e := Event()

    WWrites(if \&control then "c" else "-"
    WWrites(if \&shift then "s" else "-"
    WWrites(if \&meta then "m" else "-

    WWrite(" ", left(image(e),7),
      left("||\&x\||","||\&y\||"), 12),
      right(&interval,6), "ms")
  }
end
```

Notes:
1. \&control, et al. either succeed or fail
2. It is the act of calling Event() that causes \&x, \&control, \&interval, etc., to be set.
3. Two other values that are set: \&row and \&col
Sidebar: Reversible Drawing

By default, drawing is done in "copy" mode, which overwrites existing pixels with the pixels being drawn.

If the window attribute drawop is set to reverse, drawing a figure "inverts" the target pixels. Drawing the same figure again in the same place causes the figure to disappear, as if it had never been drawn.

The following program moves a circle across a grid.

```plaintext
procedure main() # rub1a
    WOpen("size=600,300","linewidth=3")
    every x := 50 to 550 by 50 do
        DrawLine(x, 0, x, 299)
    every y := 50 to 250 by 50 do
        DrawLine(0, y, 599, y)
    x := y := 0
    WAttrib("drawop=reverse")
    repeat {
        DrawCircle(x, y, 40) # uses defaults
        WDelay(31)           # sleeps for 31 ms
        DrawCircle(x, y, 40)
        x +:= 2
        y +:= 1
    }
end
```
Interaction example: rubberbanding

This program draws "rubberbanded" lines:

```plaintext
procedure main() # rub2
    WOpen("size=600,300","linewidth=3")
    WAttrib("drawop=reverse")
    repeat {
        case Event() of {
            &lpress: {
                start_x := &x
                start_y := &y
            }
            &ldrag: {
                DrawLine(start_x, start_y,
                        \last_x, \last_y)
                DrawLine(start_x, start_y,
                        &x, &y)
                last_x := &x
                last_y := &y
            }
            &lrelease: last_x := last_y := &null
        }
    end

Notes:

(1) A left click establishes a starting position for the line.

(2) On each drag event the previously drawn line is erased and the new line is drawn.

(3) A non-null/null value for last_x indicates that a line is/is not in progress.
Rubberbanding, continued

This slight variation draws rubberbanded circles:

```
procedure main() # rub3
   WOpen("size=600,300", "linewidth=3")
   WAttrib("drawop=reverse")
   repeat {
      case Event() of {
         &lpress: {
            start_x := &x
            start_y := &y
         }
         &ldrag: {
            r := sqrt((\last_x-start_x)^2 +
                        (last_y-start_y)^2)
            DrawCircle(start_x, start_y, \r)
            DrawCircle(start_x, start_y,
                       sqrt((&x-start_x)^2 +
                            (&y-start_y)^2))
            last_x := &x
            last_y := &y
         }
         &lrelease: last_x := r := &null
      }
   }
end
```
Interaction—blocking vs. polling

The preceding event handling examples all employ blocking—the Event() call blocks until an event is available.

An alternative to blocking is polling—the program periodically checks to see if any events are available. If so the events are processed. If not, other processing is done.

The Pending() function returns the list of events that are pending. If the list is empty, no events are pending.

Here is a version of the random point drawing program that uses polling to offer the user some control:

```plaintext
$define Height 100  # symbolic constants
$define Width 300    # via preprocessor
procedure main()    # poll
    WOpen("size=" || Width ||","||Height)
    repeat {
        if *Pending() = 0 then
            DrawPoint(?Width-1, ?Height-1)
        else
            case Event() of {
                &lpress: EraseArea(0,0,300,100)
                " ": until Event() === " 
                "Qq" : exit()
            }
    }
end
```
Example: Target game

This program draws a circular target. If the player clicks inside the target within 800ms, the radius shrinks by 10%. If not, the radius grows by 10%.

```
$define Width 600
$define Height 600
procedure main() # target
    WOpen("size="||Width||","||Height,
        "drawop=reverse")

    x := ?Width; y := ?Height; r := 50
    repeat {
        DrawCircle(x, y, r)
        hit := &null
        every 1 to 80 do {
            WDelay(10)
            while *Pending() > 0 do {
                if Event()=== &lpress then {
                    if sqrt((x-&x)^2+(y-&y)^2)
                        < r then {
                        FillCircle(x,y, r)
                        WDelay(500)
                        FillCircle(x,y,r)
                        hit := 1
                        break break
                    }
                }
            }
        }
    DrawCircle(x,y,r)
    if \hit then r *:= .9 else r *:= 1.10
    x := ?Width; y := ?Height
    } end
```