Example: Dragging objects

This program allows manipulation of randomly drawn circles.

```
record circle(x,y,r)
procedure main() # drag1
    WOpen("size=600,300","drawop=reverse")

    DrawLine(300,0,300,300)

    circles := make_circles()

    repeat case Event() of {
      &lpress:
        if c := point_in(circles, &x, &y) then {
          lastx := c.x; lasty := c.y
          r := c.r
          repeat case Event() of {
            &ldrag: {
              DrawCircle(lastx, lasty, r)
              DrawCircle(lastx := &x,
                         lasty := &y, r)
              }
            &lrelease: {
              DrawCircle(lastx, lasty, r)
              if &x <= 300 then {
                    DrawCircle(&x, &y, r)
                    c.x := &x; c.y := &y
                    }
              else
                    delete(circles, c)
                break
              }
            }
          }
        }
end
```

Example: Dragging objects, continued

Helper routines:

```
#
# Return a circle that contains the point (x,y)
#
procedure point_in(circles, x, y)
    every c := !circles do
        if sqrt((c.x-x)^2+(c.y-y)^2) < c.r then
            return c
end
#
# Create a set of randomly placed and sized
# circles
#
procedure make_circles()
    circles := set()
    every 1 to 30 do {
        r := ?40; x := ?(300-r); y := ?300
        DrawCircle(x,y,r)
        insert(circles, circle(x,y,r))
        }
    return circles
end
```

Additional behaviors to consider:

(1) Dropping one circle on another adds area to target circle.
(2) Dropping a circle on right half turns it into a square.
(3) Dropping a circle on right half adds to pile at bottom of right half.
(4) Don't center circle on pointer's hotspot.
(5) Support additional types, such as lines.
(6) Have circle pop like a bubble when dropped on right half.

Mouse tracking

There is no notion of mouse motion events in Icon's graphics system but the pointer (mouse) position can be queried via the `pointerx` and `pointery` attributes.

The following program repeatedly queries the pointer position attributes and prints the position upon a change in either coordinate:

```
procedure main() # mpoll1
    WOpen("size=300,300")
    repeat {
        x := WAttrib("pointerx")
        y := WAttrib("pointery")
        if not (x = \lastx & y = \lasty) then {
            WWrite("(", x, "," , y, ")")
            lastx := x
            lasty := y
            }

        WDelay(10)
        }
end
```

Notes:

(1) Without the `WDelay()` the CPU can be saturated.

(2) Out of window positions are reported and are relative to the upper left corner of the window.

Speculate: On a 600Mhz Windows system, how much of the CPU is consumed by the above program? How about with a smaller delay—1 millisecond?

Mouse tracking, continued

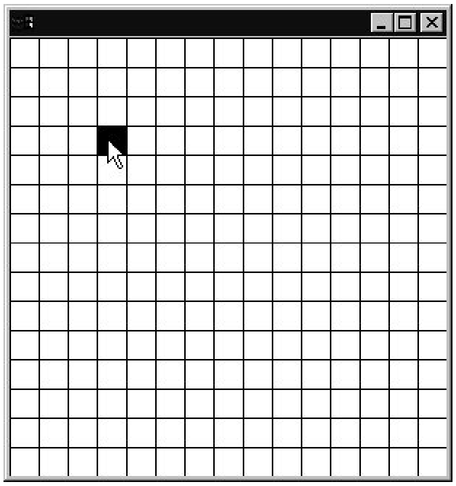
The following program tracks the pointer on a grid.

```
procedure main(args) # mpoll3
    WOpen("size=300,300")
    csize := 20
    every x := 0 to 300 by csize do
        DrawLine(x,0,x,300)
    every y := 0 to 300 by csize do
        DrawLine(0,y,300,y)

    repeat {
        x := WAttrib("pointerx") - 4
        y := WAttrib("pointery") - 23
        x := (x / csize) * csize
        y := (y / csize) * csize

        EraseArea!\last
        last := [x+1, y+1, csize-1, csize-1]
        FillRectangle!last
        WDelay(10)
        }
end
```

Notes:

(1) Note the "fudge" values of 4 and 23.

(2) Improvement: update only on pointer movement.

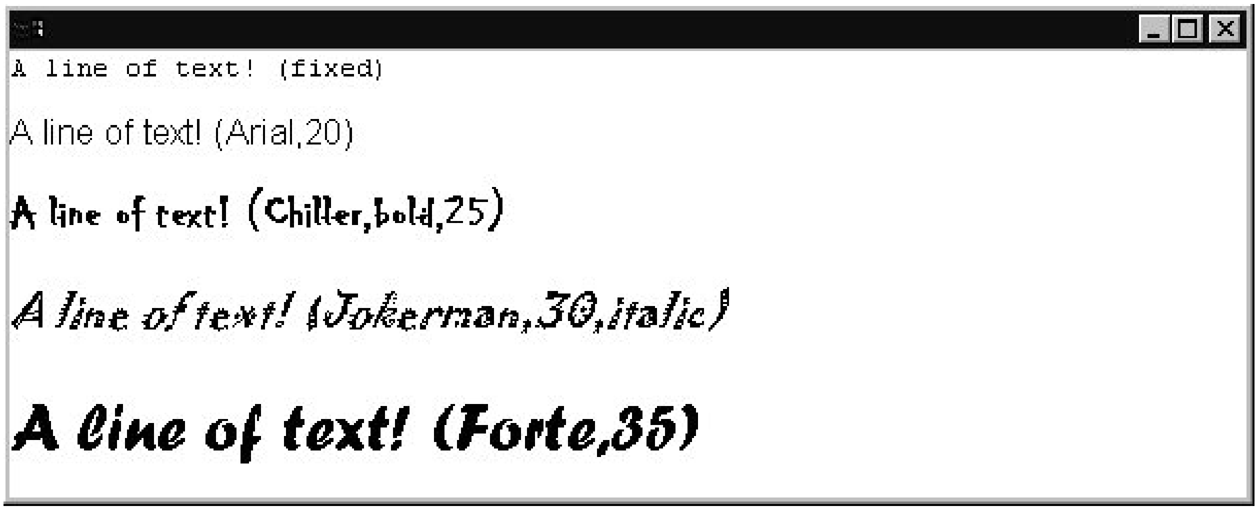


Font handling basics

One of the attributes associated with a window is its *font*. A font is a set of characters in a particular *typeface* (or *family*), *style* (such as bold or italic), and *size* (in "points").

The `font` attribute can be set or queried with `WAttrib()` or, more conveniently, with `Font()`.

```
procedure main() # font1
    WOpen("size=600,300")
    WWrite("A line of text! (", Font(), ")\n")

    specs := [
        "Arial,20", "Chiller,bold,25",
        "Jokerman,30,italic", "Forte,35"]

    every spec := !specs do {
        Font(spec)
        WWrite("A line of text! (",Font(),")\n")
        }
    WDone()
end
```



```
A line of text! (fixed)
A line of text! (Arial,20)
A line of text! (Chiller,bold,25)
A line of text! (Jokerman,30,italic)
A line of text! (Forte,35)
```

Font handling basics, continued

Typeface names are system-specific but the following names are "guaranteed" to work:

`mono`	monospaced, sans-serif
`typewriter`	monospaced, serif
`sans`	proportionally spaced, sans-serif
`serif`	proportionally spaced, serif

In a monospaced font, all characters are the same width.

Character widths vary in a proportionally spaced font.

`Font()` fails if the requested specification cannot be met.

There is no way to specify a font along with a text-output operation such as `WWrite()`. The mode of operation is always to set the `font` attribute and then perform text output operations.

Rows and columns of characters

Icon's graphics system has some support for treating a window as a two-dimensional array of characters. The involved functions assume that all characters in the window are in the same font and that the font is monospaced.

The window attributes `rows` and `columns` can be used to size a window based on rows and columns of text. The statement

```
    WOpen("font=typewriter,20", "rows=24",
           "columns=80", "cursor=on")
```

opens a window that can hold 24 rows of 80 characters of text in a 20-point monospaced font, and turns on the text cursor.

The text cursor can be positioned at a particular row and column with `GotoRC(`*`row, column`*`)`:

```
    GotoRC(10,20)
```

Two more variables that are available in conjunction with an event are `&row` and `&col`.

A start on a text editor

Here is a precursor to a text editor:

```
$include "keysyms.icn"
procedure main(args) # font2
    #
    # Read file
    every put(lines := [], !open(args[1]))
    #
    # Find length of longest line
    maxline := sort(mapf("*", lines))[-1]

    WOpen("font=typewriter,20", "cursor=on",
        "rows="||*lines+1, "columns="||maxline)
    every WWrite(!lines)

    GotoRC(1,1)
    row := col := 1

    repeat {
        case Event() of {
            Key_Down: row +:= 1  # "Arrow keys"
            Key_Up:   row -:= 1  # from keysyms.icn
            Key_Left:  col -:= 1
            Key_Right: col +:= 1
            &lpress:
                GotoRC(row := &row, col := &col)
            }
        GotoRC(*lines+1,1)
        WWrites("Row ", right(row,2),
                ", Col ", right(col,2),
               " (", (lines[row][col]|" "), ")")
        GotoRC(row,col)
        }
end
```

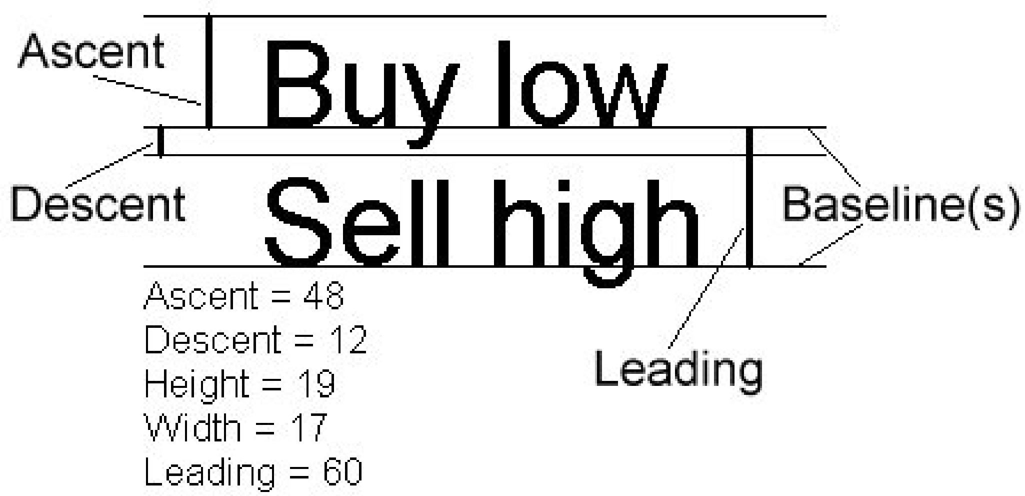
Notes:

(1) Values of `row` and `col` are not constrained.

(2) `&row` and `&col` seem misaligned on Windows.

Details on fonts

Fonts have several attributes that can be queried. These attributes are sometimes called *font metrics*.



Text is drawn so that the characters stand on a *baseline*. Some characters have *descenders* that extend below the baseline.

The *ascent* provides an amount of space above the baseline that is typically taller than the tallest character. The *descent* provides space below the baseline.

The *leading* is the space between baselines. By default it is the sum of the font's ascent and descent, but it can be set.

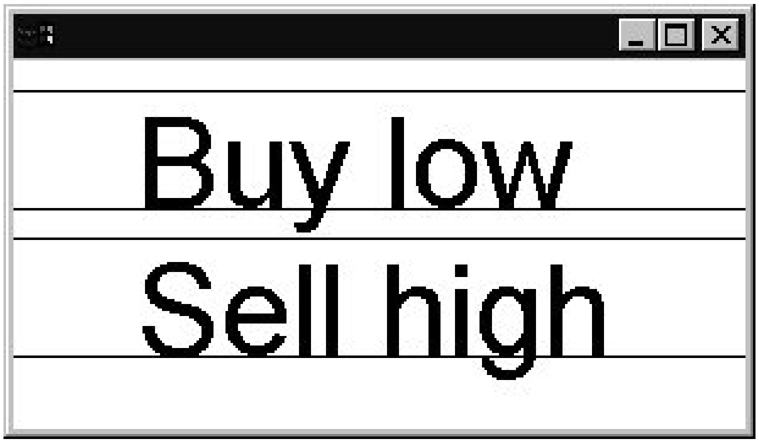
The *width* is the width of the font's widest character.

Details on fonts, continued

The routine `DrawString(x, y, s)` draws the string `s` using `y` for a baseline and positioning the left edge of the first character at `x`. Example:

```
procedure main(args) # font3
    WOpen("size=300,150","font="arial,60")
    WWrite()
    ascent := WAttrib("ascent")
    descent := WAttrib("descent")
    leading := WAttrib("leading")

    y := leading

    DrawLine(0, y, 300, y)
    DrawString(50,y, "Buy low")

    DrawLine(0, y-ascent, 300, y-ascent)
    DrawLine(0, y+descent, 300, y+descent)

    y +:= leading
    DrawLine(0, y, 300, y)
    DrawString(50,y,"Sell high")

    WDone()
end
```

Result:



Example: Boxes around text

This program reads lines from standard input and tiles the window with boxed text.

The main program reads lines and calls `drawBoxedText` to actually draw the text boxes.

Before each box is drawn the width is checked using `TextWidth(s)`, which returns the width in pixels of the string `s` when drawn in the current font.

If there is insufficient space on the current line, a new line is started by adding `leading` to `y`, and resetting `x`.

```
record box(rect, text)
global boxes
procedure main()   # font4
    boxes := set() # set of box records
    WOpen("size=600,600","font=serif,20")
    gap := 5
    x := gap
    y := 0
    while word := reverse(trim(reverse(read())))do{
        width := TextWidth(word)
        if x + width > WAttrib("width") then {
            x := gap
            y +:= WAttrib("leading") + gap
            }

        x +:= drawBoxedText(x, y, word) + gap
        }

    process(0, y)
end
```

Boxes around text, continued

The following routine displays the string `s` in a box with an upper left corner at (`x`,`y`).

```
procedure drawBoxedText(x,y,s)
    hspace := 2  # pad with two pixels
    width := TextWidth(s) + hspace*2
    ascent := WAttrib("ascent")
    descent := WAttrib("descent")
    baseline := y + ascent
    height := ascent + descent

    DrawString(x+hspace, baseline, s)

    rect := [x,y,width,height]
    DrawRectangle!rect

    insert(boxes, box(rect,s))
    return width
end
```

The following routine uses `GotoXY()` to position the text cursor and then processes events, using `WWrite()` to print words that are clicked on.

```
procedure process(x, y)
    Font(Font()||",italic")
    GotoXY(x,y + WAttrib("leading") * 2)
    repeat case Event() of {
        &lpress: {
          every b := !boxes do {
            rect := b.rect
            if rect[1] <= &x <= rect[1]+rect[3] &
              rect[2] <= &y <= rect[2]+rect[4] then
                 WWrite(b.text)
                }
            }
        }
end
```