Multiple Windows

Icon's graphics system supports multiple windows.

\texttt{WOpen()} returns a value of type \texttt{window}. A side effect of the first call to \texttt{WOpen()} is that the resulting value is assigned to \texttt{&window} (the \textit{subject window}).

Almost every graphics procedure accepts a window as its first argument. Examples:

\begin{verbatim}
  DrawPoint(W, x, y)
  Font(W, s)
  WWrite(W, s1, s2, ...)
\end{verbatim}

If the first argument to a graphics procedure is not of type \texttt{window}, \texttt{&window} is assumed as an implicit first argument.

This program,

\begin{verbatim}
  procedure main()
    WOpen("size=300,400")
    WWrite("Hello, world!")
    WDone()
  end
\end{verbatim}

and this program,

\begin{verbatim}
  main()
    w := WOpen("size=300,400")
    WWrite(w, "Hello, world!")
    WDone(w)
  end
\end{verbatim}

are equivalent.
Multiple windows, continued

This program creates four windows, using the `pos` attribute to position the first three windows. The fourth window prints a count of events received in the other three.

```plaintext
procedure main(args) # mwin1
    sz := "size=200,200"
    w1 := WOpen(sz, "label=One", "pos=300,0")
    w2 := WOpen(sz, "label=Two", "pos=100,300")
    w3 := WOpen(sz, "label=Three", "pos=500,300")

    wins := [w1, w2, w3]
    events := table(0)
    &window :=
        WOpen("size=200,300","font=typewriter,25")

    repeat every w := !wins do {
        if *Pending(w) > 0 then {
            WWrite(w, Event(w))
            events[w] +:= 1
            EraseArea()
            GotoRC(1,1)
            every p := !sort(events,2) do
                WWrite(left(WAttrib(p[1],"label"),10),p[2])
        }
    }
end
```

An alternative to polling with `Pending()` is to use `Active()`, which returns a window that has an event pending, blocking if there are none.

```plaintext
repeat {
    w := Active()
    WWrite(w, Event(w))
    ...
```
Multiple windows, continued

`Raise(W)` causes the window `W` to be brought to the top of the window stack, so that no other window obscures it. Raising a window typically causes it to become the active window.

The following program makes five overlapping windows and then raises windows as indicated by the user.

```plaintext
procedure main() # mwin2
    WOpen("size=400,300")
    wins := []
    every i := 1 to 5 do {
        put(wins, WOpen("label=Window "||i,
            "size=200,200",
            "pos=500,"||i*20))
    }
    Raise(&window)
    repeat {
        WWrites("Window? ")
        win := WRead()
        Raise(wins[integer(win)])
    }
    Raise(&window) # without this the raised window would retain # the focus
end
```

There is a counterpart procedure, `Lower(W)`.

A window can be closed with `WClose(W)`. If the subject window is closed, `&window` is set to null.
Windows, canvases, and graphics contexts

A window is actually a coupling between a canvas and a graphics context. Think of it this way:

    record window(canvas, graphics_context)

The canvas represents the on-screen artifact. Drawing operations change pixels on the canvas.

The graphics context holds a collection of information that is used to control drawing on the canvas.

Each window attribute is actually associated with either the canvas or the graphics context. Here's a partial list based on the attributes that we've covered:

Attributes associated with the graphics context:
  bg, fg, drawop, linewidth, dx, dy, font-related attributes (font, fheight, leading, etc.), clipping region

Attributes associated with the canvas:
  Dimensions (width, rows, etc.), label, pos, row, col, pointerx, pointery

See Appendix G in the text for a complete list.
Windows, canvases, and GCs, continued

This statement:

\[
\text{w := WOpen("size=300,300","label=MyWin", "linewidth=9")}
\]

Creates this coupling:

The various graphics procedures use information from the canvas and/or the graphics context to perform the desired operations.
Multiple graphics contexts for a canvas

In some cases there's a need to regularly change graphics context attributes, perhaps toggling between two settings for color, but it's tedious and error-prone to make regular changes with WAttrib().

A better alternative is provided by cloning, which produces a window that couples a new graphics context with an existing canvas.

Given this coupling:

```
Canvas
size=300,300
label=MyWin
```

```
Window w
```

```
Graphics
Context
fg=black
linewidth=9
```

the statement

```
    w2 := Clone(w, "fg=red", "linewidth=21")
```

produces this:

```
Canvas
size=300,300
label=MyWin
```

```
Window w
```

```
Graphics
Context
fg=black
linewidth=9
```

```
Window w2
```

```
Graphics
Context
fg=red
linewidth=21
```

Non-overridden graphics context attributes are copied from $w$. 
Multiple GCs for a canvas, continued

For reference:

A line drawn using window \( w \) will be black and 9 pixels wide.

A line drawn using window \( w_2 \) will be red and 21 pixels wide.

Example:

```plaintext
procedure main() # clone1
    w := WOpen("size=300,300","label=MyWin",
                "linewidth=9")
    w2 := Clone(w, "fg=red", "linewidth=21")

    every x := 0 to 300 by 50 do
        every DrawLine((w2|w),x,0,x,300)

    WDone()
end
```

Note that the thicker line must be drawn first to achieve the desired effect.
Multiple GCs for a canvas, continued

This program uses translation, clipping, and cloning to "echo" points on the left half of the window with circles on the right half.

```cpp
procedure main() # clone
left := WOpen("size=600,300")
#
# Constrain drawing to left half of window
Clip(left, 0, 0, 300, 300)
#
# Establish two new graphics contexts, both
# with X-coordinate translation and one with
# a pale red foreground color
right := Clone(left, "dx=300","fg=pale red")
right2 := Clone(left, "dx=300")
#
# Constrain the echoes to the right half
Clip(right, 0, 0, 300, 300)
Clip(right2, 0, 0, 300, 300)

Height := Width := 300
while e := Event(left) do {
    case e of {
        &lpress|&ldrag: {
            DrawPoint(left, &x, &y)
            FillCircle(right, &x, &y, 10)
            FillCircle(right2, &x, &y, 5)
        }
    }
}
```

What would this program be like without using translation, clipping and cloning?