Icon Graphics—Introduction

Facilities for graphical programming in Icon evolved in the period 1990-1994.

A philosophy of Icon is to insulate the programmer from details and place the burden on the language implementation. The graphics facilities were designed with same philosophy.

Icon's graphical facilities are built on the X Window System on UNIX machines. On Microsoft Windows platforms the facilities on built on the Windows API.

Window basics

Before any graphical operations can be done, a window must be opened.

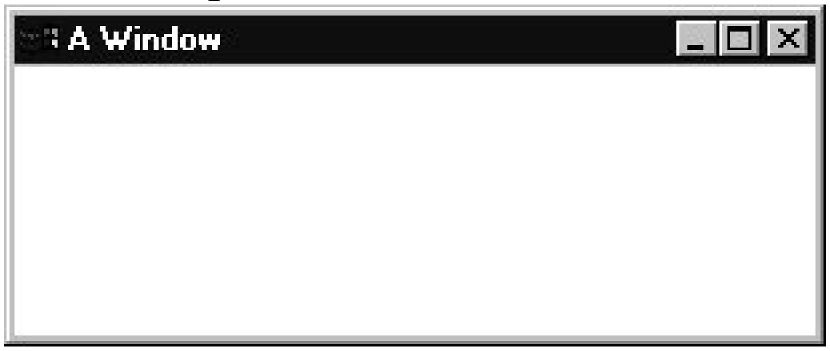
Here is a complete program that opens a window with a specific width, height, and label:

```
link graphics
procedure main() # win1
    WOpen("height=100","width=300",
        "label=A Window")
    WDone()
end
```

As a rule, graphics programs should `link graphics`.

On UNIX the program can be compiled with `icont`, as usual. Use `wicont` on Windows.

On a Windows platform, here's the result:



`WOpen()` accepts zero or more window *attributes* as arguments. Attributes may be specified in any order.

`WDone()` waits until a `q` or `Q` is typed in the window.

Window basics, continued

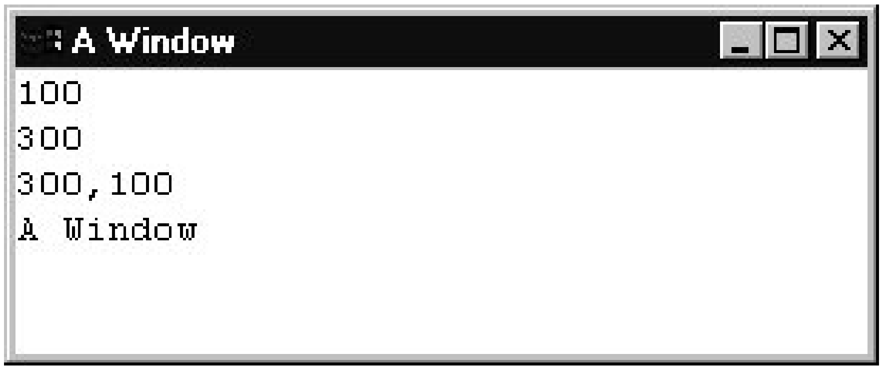
Window attributes can be queried with `WAttrib(s1, s2, ...)`. The value of each named attribute is generated.

`WWRite()` is like `write()`, but sends output to the window.

Example:

```
link graphics
procedure main() # win2
    WOpen("height=100","width=300",
        "label=A Window")
    every WWrite(WAttrib("height", "width",
                         "size", "label"))
    WDone()
end
```

Resulting window:



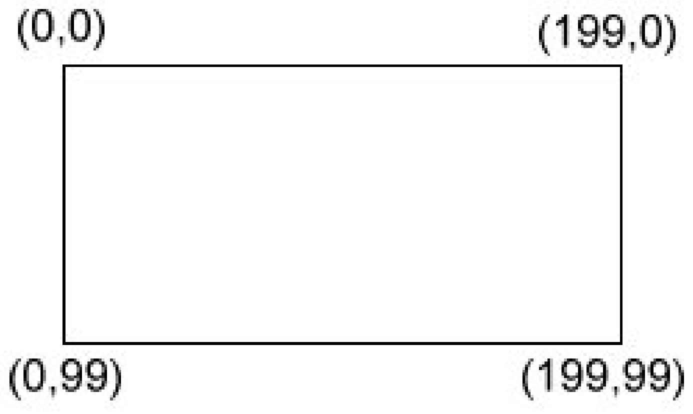
```
100
300
300,100
A Window
```

In essence, `WWrite()` treats the window as a scrolling text window.

`write()` could be used instead of `WWrite()`; output would then go to the "console".

Coordinate system

The coordinate system is integer based with (0,0) in the upper left corner of the window. Here are the corner points for a window with `size=200,100`:



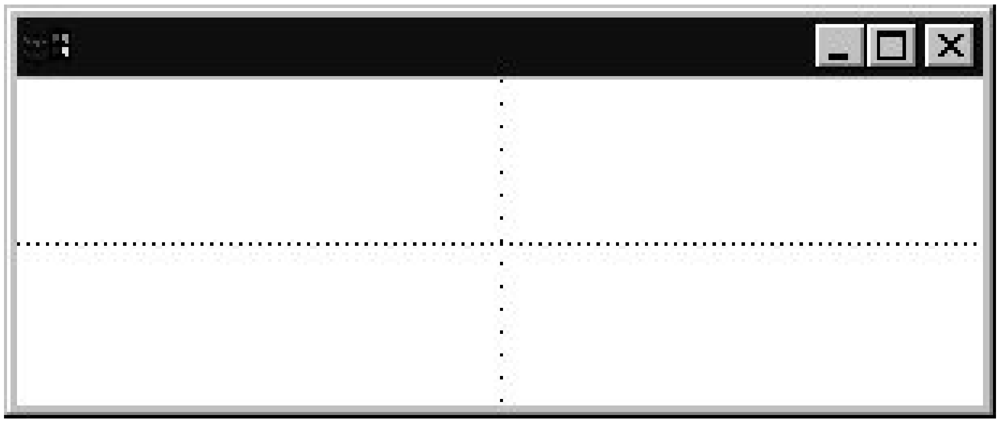
Drawing points

The simplest drawing primitive is `DrawPoint(x, y)`, which draws one pixel at the specified coordinates in the foreground color (black, by default).

Example:

```
link graphics
procedure main() # dp1
    WOpen("size=300,100")

    every x := 0 to 299 by 3 do
        DrawPoint(x, 50)    # horizontal

    every y := 0 to 99 by 7 do
        DrawPoint(150, y)   # vertical

    WDone()
end
```

Result:



Drawing points, continued

Some fun with randomly drawn points:

```
link graphics

$define Height 100  # symbolic constants
$define Width 300   #  via preprocessor

procedure main() # dp2
    WOpen("size=" || Width ||","||Height)

    repeat {
        DrawPoint(?Width-1, ?Height-1)
        }

    WDone()
end
```

Another angle:

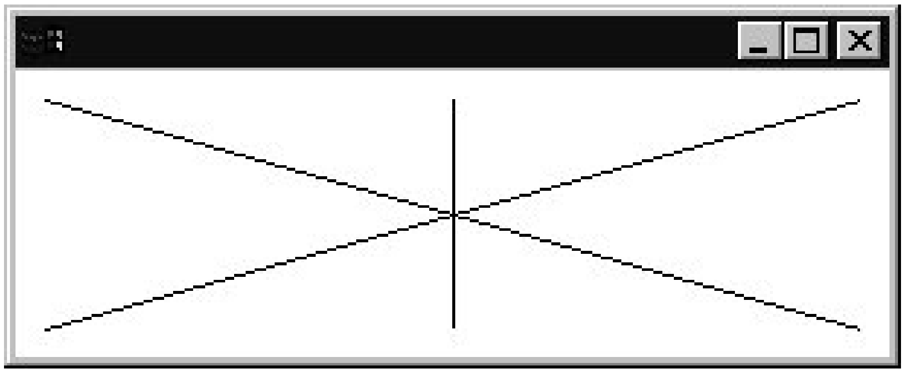
```
link graphics
$define Height 100
$define Width 300

procedure main(args) # dp3
    WOpen("size=" || Width ||","||Height)
    N := args[1] | 1

    repeat {
        x := y := 0
        every 1 to N do x +:= ?(Width/N)
        every 1 to N do y +:= ?(Height/N)
        DrawPoint(x,y)
        }

    WDone()
end
```

Drawing lines

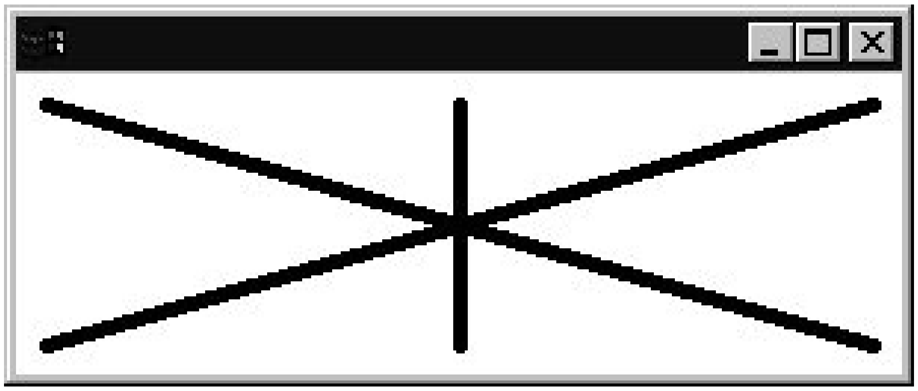
`DrawLine(x1, y1, x2, y2)` draws a line between the points (`x1`, `y1`) and (`x2`, `y2`), inclusive.

```
link graphics
procedure main(args) # dl2
    WOpen("size=300,100")
    WAttrib("linewidth=" || args[1])
    DrawLine(10, 10, 290, 90)
    DrawLine(10, 90, 290, 10)
    DrawLine(150, 10, 150, 90)
    WDone()
end
```

When run with no arguments, a default `linewidth` of 1 is used:



Here is a `linewidth` of 5:

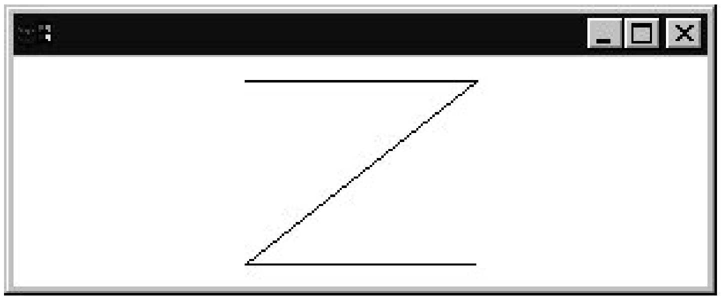


Drawing lines, continued

An arbitrary number of coordinate pairs can be passed to `DrawLine`. It draws a line between the first and second points, then the second and third points, etc.

```
procedure main() # dl3
    WOpen("size=300,100")
    DrawLine(100,10,200,10,100,90,200,90)
    WDone()
end
```

Result:



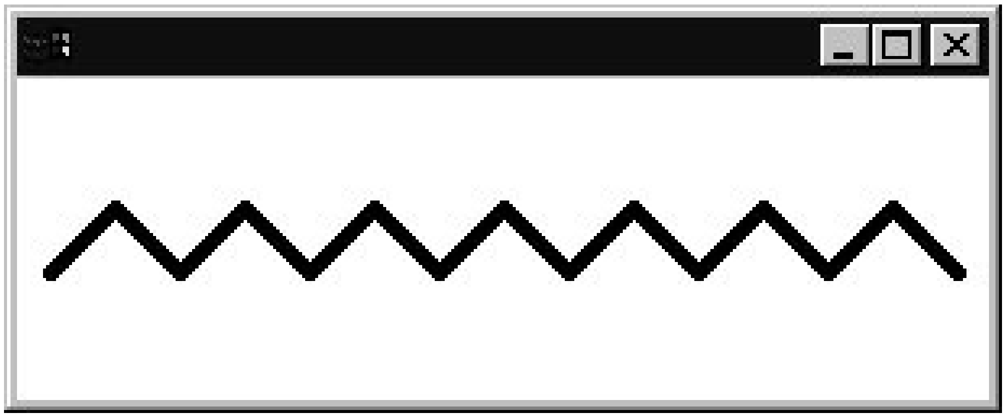
Icon's *list invocation* syntax is often used with drawing functions that accept a variable number of arguments:

```
procedure main() # dl3a
    WOpen("size=300,100")

    zpts := [100,10,200,10,100,90,200,90]
    DrawLine!zpts  # "list invocation"
    WDone()
end
```

A related function is `DrawSegment`, which draws disjoint segments for each pair of coordinate pairs.

Drawing lines, continued

Problem: Write a program that produces an approximation of this image:



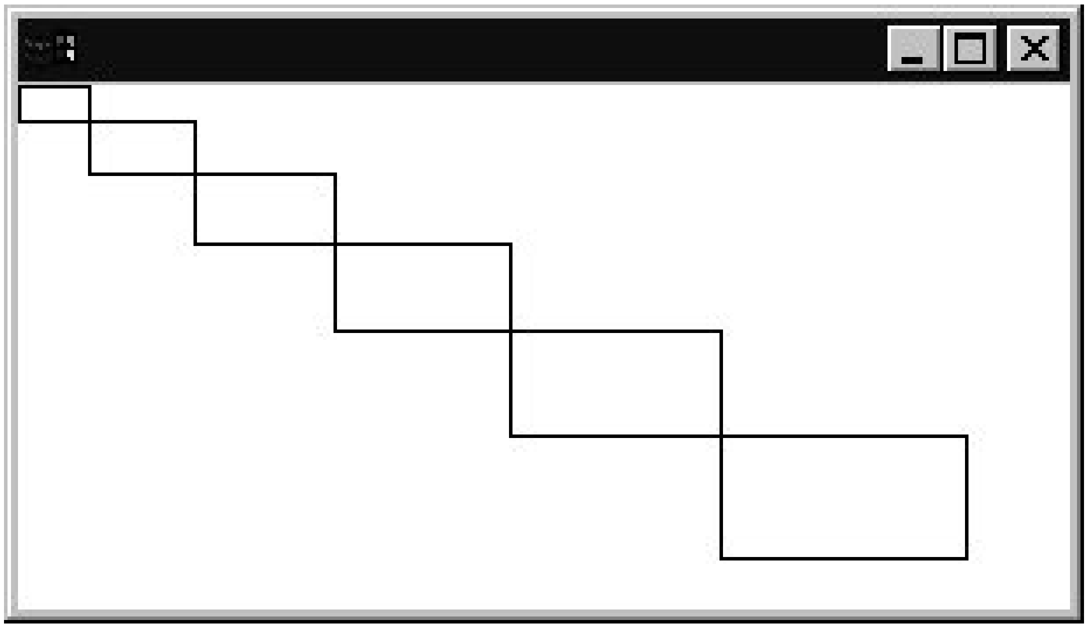
Drawing rectangles

The function `DrawRectangle(x, y, w, h)` draws the outline of a rectangle.

With a line width of 1, the upper left corner is at (`x`, `y`) and the lower right corner is at (`x+w`, `y+h`).

```
procedure main(args) # dr1
    WOpen("size=300,150")
    x := y := 0
    every h := 10 to 35 by 5 do {
        DrawRectangle(x, y, h*2, h)
        x +:= h*2
        y +:= h
        }
    WDone()
end
```

Result:



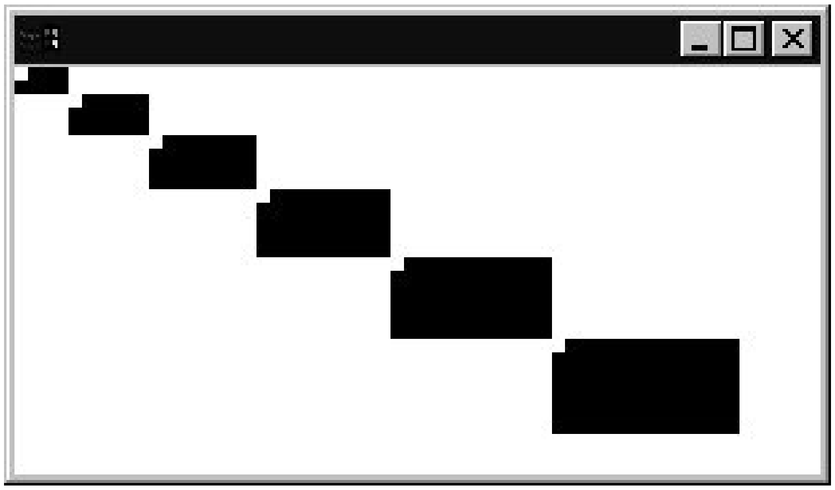
Drawing rectangles, continued

`FillRectangle(x, y, w, h)` is just like `DrawRectangle` but it produces a rectangle filled with the foreground color.

A related function is `EraseArea`, which accepts the same arguments and fills the rectangular area with the background color (white, by default).

```
procedure main(args) # dr2
    WOpen("size=300,150")
    x := y := 0
    every h := 10 to 35 by 5 do {
        FillRectangle(x, y, h*2, h)
        EraseArea(x, y, 5, 5)
        x +:= h*2
        y +:= h
        }
    WDone()
end
```

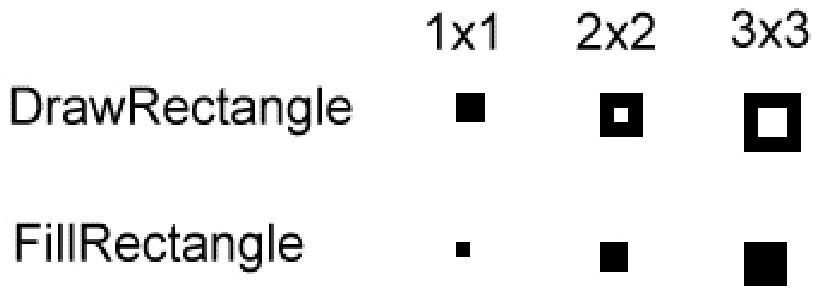
Result:



Drawing rectangles, continued

Appendix I in the text covers some painful but important details about the rendering of various figures.

One example of "interesting" behavior is the difference in the rectangular area when drawn with `DrawRectangle` versus `FillRectangle`:

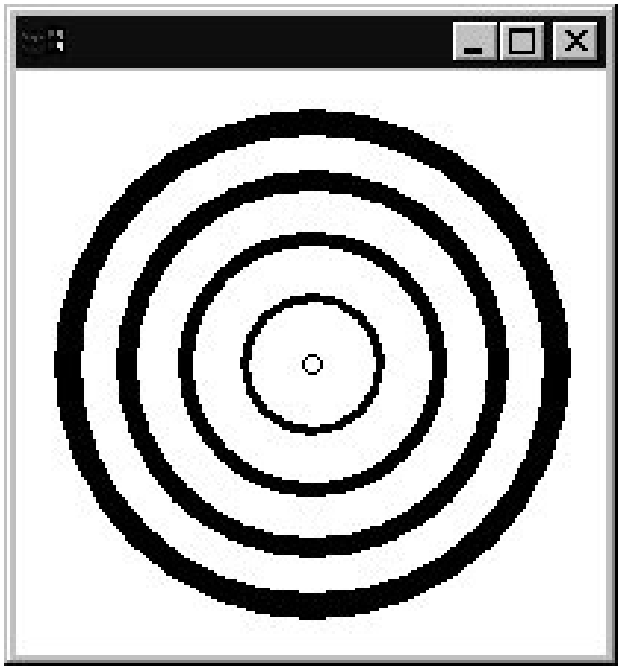


Drawing circles

Circles are drawn with `DrawCircle(x, y, radius)`:

```
procedure main(args) # dc1
    WOpen("size=200,200")

    width := 1
    every r := 3 to 100 by 20 do {
        DrawCircle(100, 100, r)
        WAttrib("linewidth=" || (width +:= 2))
        }

    WDone()
end
```

Result:



Drawing circles, continued

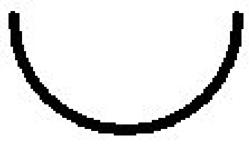
The previous example used some defaults. `DrawCircle` is actually more general:

```
    DrawCircle(x, y, r, start, extent)
```

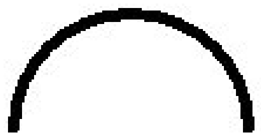
This draws a circular arc centered at (`x`, `y`) with radius `r` starting at `start` radians and continuing through `extent` radians. (Recall that 2π radians equals 360 degrees.)

`start` is measured with zero at 3 o'clock. Positive values for start and extent indicate a clockwise direction; negative values indicate counter-clockwise direction.

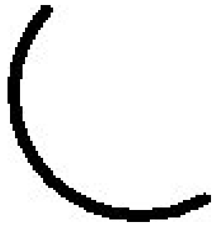
DrawCircle(..., 0, &pi)

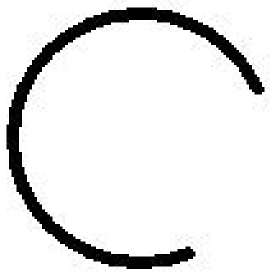


DrawCircle(..., &pi, &pi)



DrawCircle(..., &pi/3, &pi*.9)





DrawCircle(..., -&pi/8, -&pi*3/2)

Drawing circles, continued

Here is a simple-minded test program that exercises `DrawCircle` and its counterpart, `FillCircle`:

```
procedure main(args) # dc3
    WOpen("size=200,240",
        "linewidth=10")

    WWrite(repl("\n",30))
    repeat {
        EraseArea()

        WWrite("f/d, start, extent? ")

        args := split(WRead())
        p := if get(args) == "f"
             then FillCircle else DrawCircle

        every !args *:= (2*&pi)/360

        p!([100,100,90]|||args)
        WRead()
        }
end
```

Notes:

Via defaults, `EraseArea()` erases the entire window.

`WRead()` reads a line of input typed directly into the window.

Drawing arcs

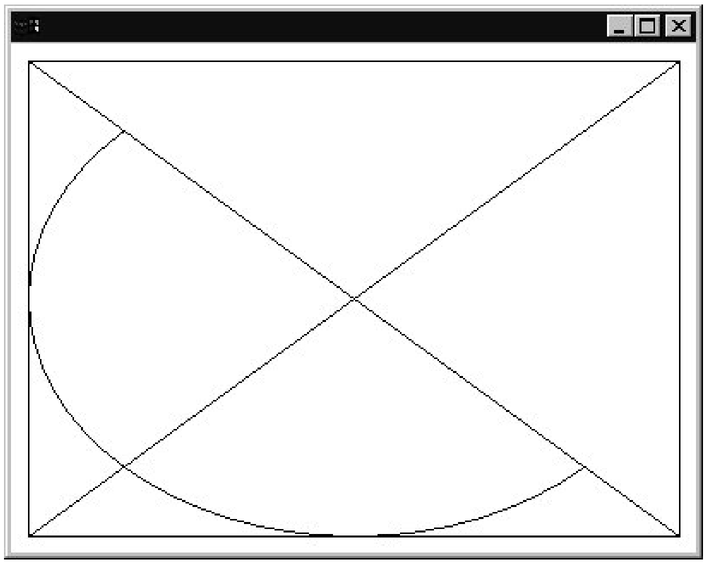
`DrawArc(x, y, width, height, start, extent)` draws an arc that is "inscribed" in the rectangle specified by the first four parameters.

`start` and `extent` specify the starting position and angular extent of the figure, just like `DrawCircle`.

Here is an example from the text, page 84:

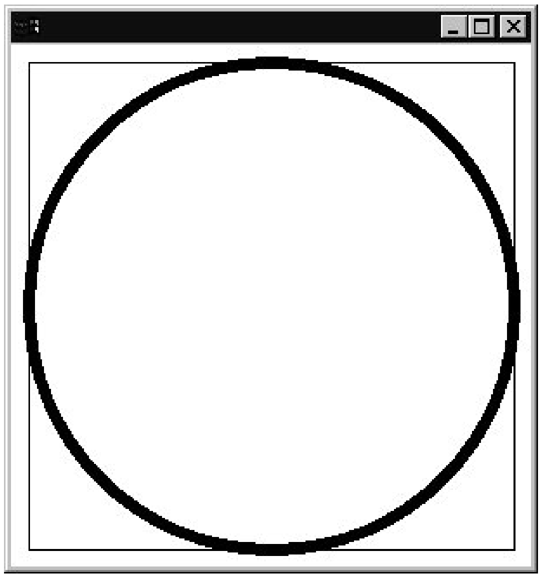
```
procedure main() # arc1, from GPiI, p.84
    WOpen("size=400,300")

    DrawRectangle(10, 10, 380, 280)

    DrawLine(10,10, 390, 290)
    DrawLine(10, 290, 390, 10)

    DrawArc(10, 10, 380, 280, &pi/4, &pi)
    WDone()
end
```

Result:



Drawing arcs, continued

Inscribing 2π arc in a square produces a circle:



Note that the thick stroke is centered on the bounding rectangle. Here's the code:

```
procedure main(args) # arc2
    WOpen("size=300,300")

    DrawRectangle(10, 10, 280, 280)

    WAttrib("linewidth=7")

    DrawArc(10, 10, 280, 280, 0, &pi*2)

    WDone()
end
```

Mnemonic aid for the order of `start` and `extent`: "The start comes first."

Sidebar: The `case` statement

Icon's `case` statement provides for execution of a block of code based on a discriminating value, much like `switch` in Java.

A simple example:

```
procedure main()

    every i := ![2,1,3,4] do {

        case i of {
            1: { write("first") }
            2: { write("second") }
            3: { write("third") }
            default: {
                write("other")
                notify_support(i)
                }
            }
        }
end
```

Output:

```
second
first
third
other
```

Note that the element following the colon is an expression. In the above example the braces are optional in the first three case clauses.

The `default` clause is optional. If omitted and no value matches, the statement fails.

The `case` statement, continued

Note that the case selectors do not need to be constants:

```
procedure main()
    writes("x? ")
    x := read()
    writes("y? ")
    y := read()

    while line := read() do {
        case line of {
            x: write("Looks like an x...")
            y: write("It's a y!")
            default: write("Hmm...")
            }
        }
end
```

Interaction:

```
x? 3
y? 7
1
Hmm...
3
Looks like an x...
6
Hmm...
7
It's a y!
10
Hmm...
```

The `case` statement, continued

A selector may be an arbitrary expression, and be generative:

```
procedure main()
    every c := !read() do {
        what := case c of {
            !&lcase:      "L"
            !&ucase:      "U"
            !&digits:     "D"
            "."|","|"?":  "P"
            whitespace(): "W"
            "\n": c
            default: "?"
            }
        writes(what)
        }
    write()
end

procedure whitespace()
    suspend !" \t"
end
```

Interaction:

```
Line?
Test #3??    (input)
ULLLW?DPP
```

Note that selection is done using exact equality (===).

```
][ case 1 of { "1": "yes" };
Failure

][ case 1 of { 1: "yes" };
   r := "yes"  (string)
```

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:

```
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:

```
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:

```
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 `Event()` is a string that corresponds to the key. For other keys, such as the function keys and cursor keys, `Event()` produces an integer.

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

The library file `keysyms.icn` has `$defines` for various non-textual keys. Examples:

```
$define Key_Home                36
$define Key_Insert              45
$define Key_F1                  112
```

Use `$include "keysyms.icn"` (not `link`).

Keystrokes and mouse actions can be intermixed:

```
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:

```
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`

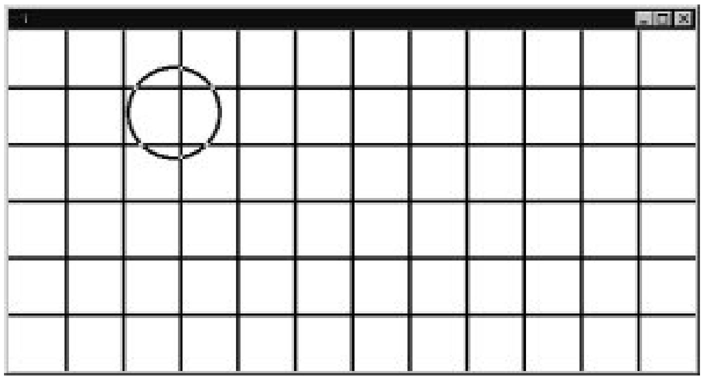
Sidebar: Reversible Drawing

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.

```
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:

```
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.

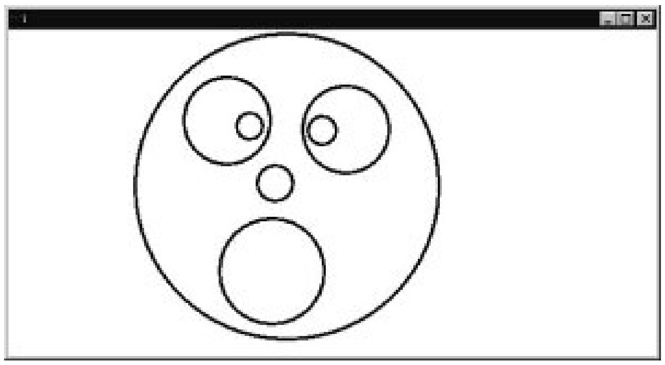
(2) On each drag event the previously drawn line is erased and the new line is drawn.

(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:

```
$define Height 100  # symbolic constants
$define Width 300   #  via preprocessor
procedure main() # poll1
    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
```

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

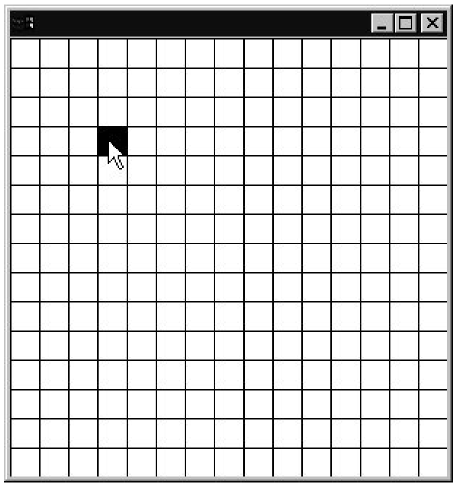
The following program tracks the pointer on a grid.

```
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.

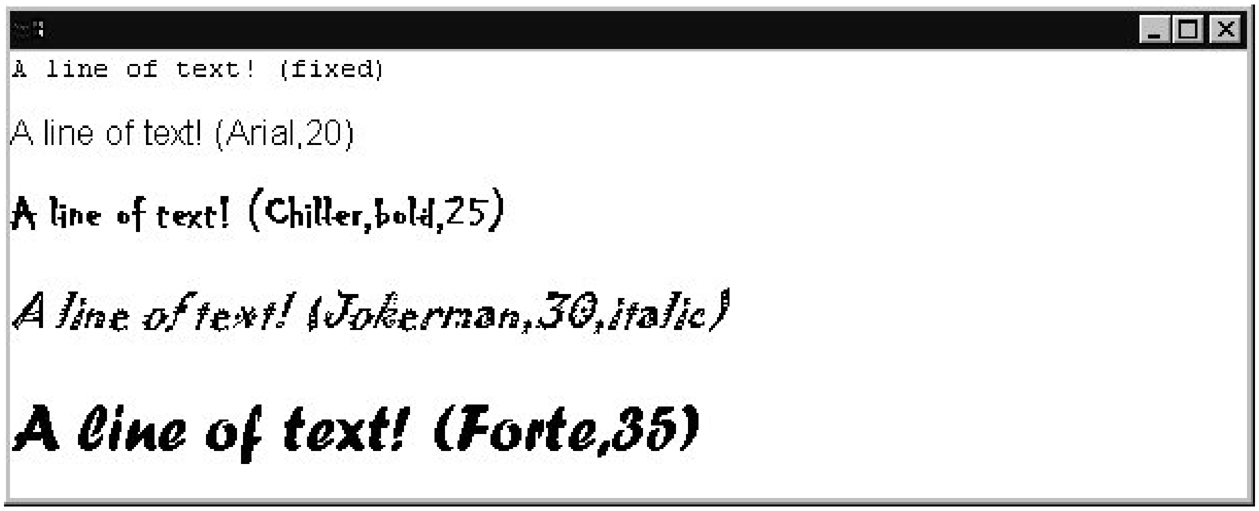


Font handling basics

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
```

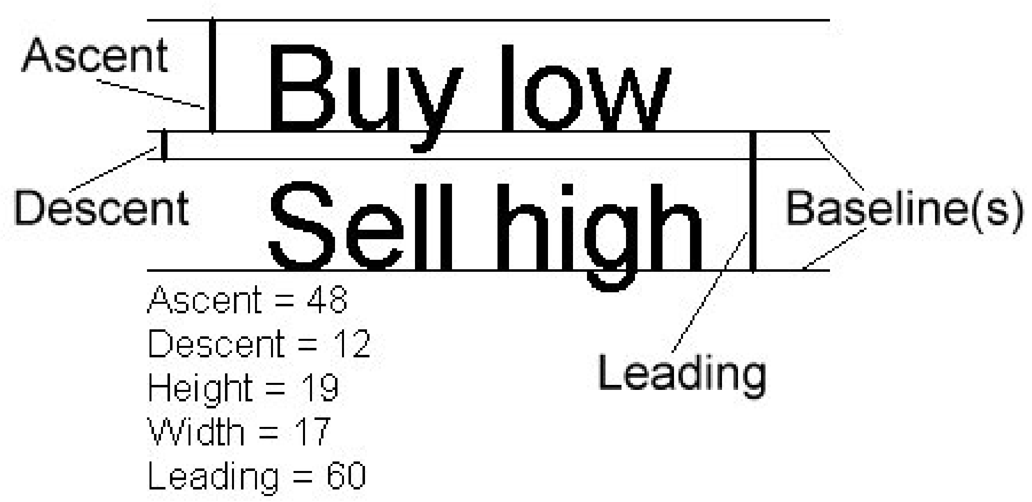
Notes:

(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.

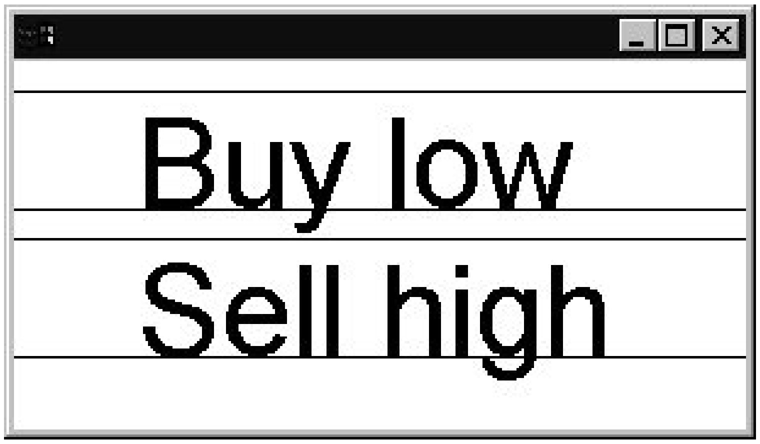
The *width* is the width of the font's widest character.

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
```

`DrawString` vs. `WWrite` et al.:

`WWrite()` and `WWrites()` produce output at the current position of the text cursor and appropriately update the position of the text cursor.

The text cursor's position can be set with `GotoRC()` and `GotoXY()`. Its position can be queried via the attributes `x` and `y` (coordinates) and `row` and `col`.

`DrawString()` produces output at the specified position and does not update the text cursor.

`DrawString()` changes only the pixels of the characters; `WWrite()` outputs a rectangle of pixels.

`DrawString()`, in conjunction with `drawop=reverse`, can be used to animate text. (But this does not work on Windows.)

Bottom line:

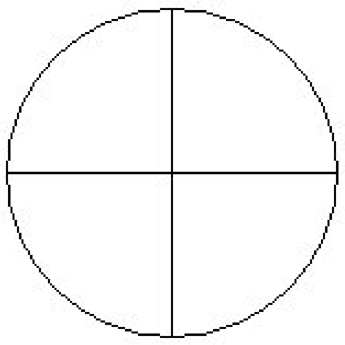
> `WWrite()` is convenient, especially with monospaced text.
>
> `DrawString()` provides full control.

`DrawString/TextWidth` and `GotoXY/WWrites` are roughly equal "teams".

Coordinate translation

The dx and dy attributes specify a *translation* of the X and Y coordinates. If dx and/or dy have a non-zero value the value is automatically added to the X and/or Y coordinate specified in subsequent graphics calls.

Consider this figure:



Here is code to draw it centered at (100,00) with a radius of 75:

```
x :=  y := 100
r := 75
DrawCircle(x, y, r)
DrawSegment(x-r, y, x+r, y, x, y-r, x, y+r)
```

Here is code that uses translation:

```
WAttrib("dx=100","dy=100")
r := 75
DrawCircle(0, 0, r)
DrawSegment(-r, 0, r, 0, 0, -r, 0, r)
```

Changes to dx and dy are <u>not</u> cumulative.

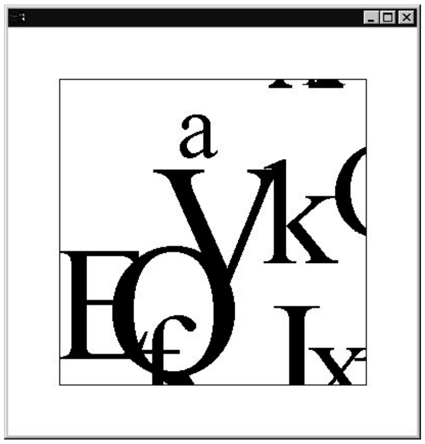
Clipping

Graphics libraries and/or host operating systems typically constrain graphical output to the target window—if a figure extends beyond the bounds of the window the out of bounds pixels are simply not drawn.

In some cases it is desirable to limit drawing to a portion of a window. The procedure `Clip(x, y, w, h)` sets a *clipping region*—no pixels will be drawn outside the specified rectangle.

The following program draws randomly sized characters at random positions on the screen. A clipping region is used to constrain the output to the center of the window.

```
procedure main() # clip1
    WOpen("size=400,400")
    center_square := [50,50,300,300]
    DrawRectangle!center_square
    Clip!center_square
    repeat {
        Font("serif,"||(60+?200)) | stop()
        DrawString(?400, ?400, ?&letters)
        if *Pending() > 0 then
            Event() & Event()
        WDelay(70)
        }
end
```



Example: Clipping and translation

This program draws random circles. A square clipping region is initially established at the center of the window and gradually increased.

When the clipping region reaches the full size of the window, the foreground and background colors are reversed (via the `reverse` attribute), the window is erased, and the process repeats.

Coordinate translation is used both for drawing and defining the clipping region.

```
procedure main() # clip2
    WOpen("size=400,400","dx=200","dy=200")

    rev := create |!["on","off"]
    side := 400
    repeat {

        every i := 1 to side by 5 do {
            WAttrib("dx="||200-i/2,
                    "dy="||200-i/2)

            Clip(0,0,i,i)
            every 1 to 20 do
                DrawCircle(?i, ?i, ?25)

            if *Pending() > 0 then
                Event() & Event()
            WDelay(70)
            }
        WAttrib("reverse="||@rev)
        EraseArea()
        }
end
```

Color specification

A window has attributes for the foreground and background colors (`fg` and `bg`). They can be set via `WAttrib()` or with the `Fg(s)` and `Bg(s)` procedures.

Routines such as `DrawCircle` and `FillRectangle` draw pixels in the foreground color, which is `black` by default.

A simple way to specify a color is by naming one of these *hues*:

```
black       orange
gray        yellow
white       green
pink        cyan
violet      blue
brown       purple
red         magenta
```

One way to think of hue: The basic nature of a color.

Example:

```
procedure main() # color1
    WOpen("size=300,300")
    colors := split("black gray white pink _
       violet brown red orange yellow green _
       cyan blue purple magenta")

    every color := !colors do {
        Bg(color)
        EraseArea()
        until Event() === &lpress
        }
end
```

Color specification, continued

Icon's color naming system was inspired by a 1982 paper by Berk, et al.: *A New Color-Naming System for Graphics Languages* that uses natural language to describe a color. Here is the full form:

lightness	saturation	*hue1*	*hue2*

`pale`
`light`
`medium`
`dark`
`deep`
	`weak`
	`moderate`
	`strong`
	`vivid`
		`hue``[ish]`	*`hue`*

Saturation is a measure of how far the color is from a gray.

Lightness is the intensity of a color.

Examples:

```
pale green
pale weak green
yellow green
greenish yellow
pale greenish yellow
moderate pinkish red
dark bluish purple
```

All elements are optional except for *hue2*. The defaults of `medium` and `vivid` are underlined.

A specification like "`yellow orange`" selects a color halfway between yellow and orange. "`yellowish orange`" specifies a color 3/4 of the way toward orange.

Color specification, continued

The `colrbook` program in the IPL displays a hue with varying levels of lightness and saturation.

Here's a simple program for testing color specifications:

```
procedure main() # color2
    WOpen("size=300,600")

    WAttrib("font=serif,30")
    WWrite()

    y := WAttrib("fheight")
    striph := 75

    while GotoRC(1,1) &
          WWrites(repl(" ",100),"\r") &
          color := WRead() do {
        if *color = 0 then { # <Enter> clears
            EraseArea()
            y := WAttrib("fheight")
            next
            }
        Fg(color) | next
        FillRectangle(0,y,300,striph)
        Fg("black")
        DrawString(10,y+striph/2,color)
        y +:= striph
        }
end
```

Numerical color specification

A color can also be specified numerically, in terms of the brightness of red, green, and blue light. One form is a comma-separated triple of decimal integer values in the range 0 to 65,535:

```
    <red>,<green>,<blue>
```

Examples:

```
    Fg("60000,0,0")      # bright red
    Fg("0,0,30000")      # fairly dark blue
    Bg("50000,50000,50000") # light gray
    Fg("40000,30000,50000") # pale purple
```

Zero for all three yields black; maximum values yield white.

Alternatively, values can be specified using triples of 1-4 hex digits:

```
    Fg("#f00")
    Bg("#ff21a")
    Fg("#7ffa00b88")
    Bg("#123456789abc")
```

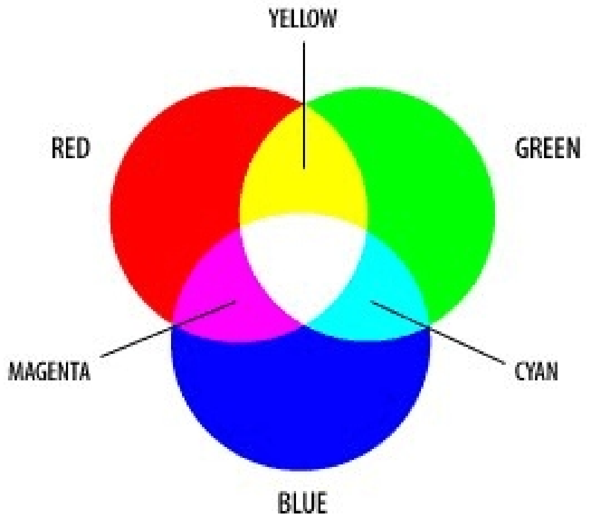
With the hexademical form the number of digits must be a multiple of three.

The procedure `ColorValue(s)` produces a string that is the decimal triple form of the color named by the string `s`.

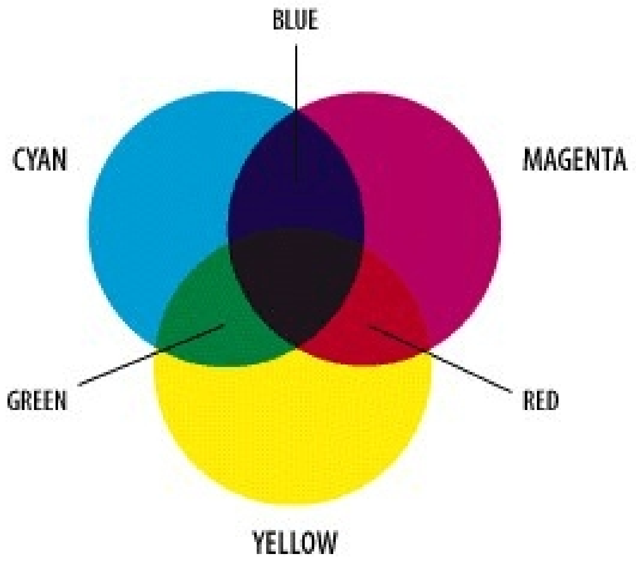
The sample program `color2a` is simply `color2` augmented to show the result of `ColorValue()`.

Color models

The RGB color model is *additive*—light from three different component colors contribute to the final value.



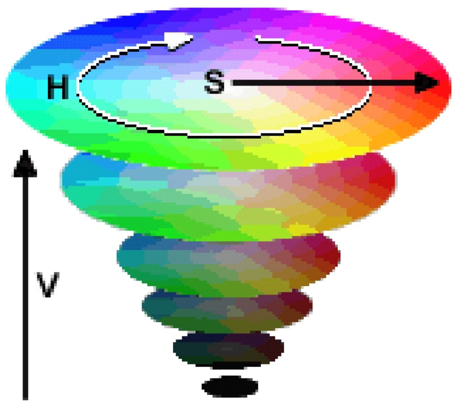
The CMY color model is commonly used when printing colors. It is called a subtractive model because ink is used to subtract colors from the image. The colors cyan, magenta, and yellow reflect no red, green, or blue light, respectively.



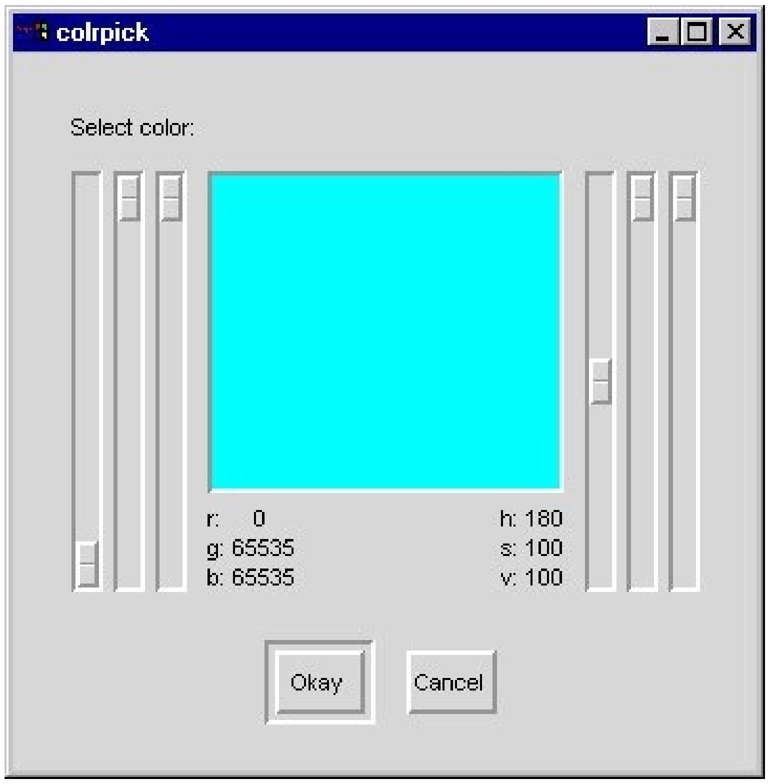
Diagrams from Adobe.com

Color models, continued

A third color model is HSV (Hue, Saturation, Value). "Value" is the brightness of the color. Here is a conical view of the HSV space from `www.wikipedia.org`:



The IPL program `colrpick` can be used to see the correspondence between the RGB and HSV models:



```
r:     0            h: 180
g: 65535            s: 100
b: 65535            v: 100
```

Multiple Windows

Icon's graphics system supports multiple windows.

`WOpen()` returns a value of type `window`. A side effect of the first call to `WOpen()` is that the resulting value is assigned to `&window` (the *subject window*).

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

```
DrawPoint(W, x, y)
Font(W, s)
WWrite(W, s1, s2, ...)
```

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

This program,

```
procedure main()
    WOpen("size=300,400")
    WWrite("Hello, world!")
    WDone()
end
```

and this program,

```
main()
    w := WOpen("size=300,400")
    WWrite(w, "Hello, world!")
    WDone(w)
end
```

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.

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

```
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.

```
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`

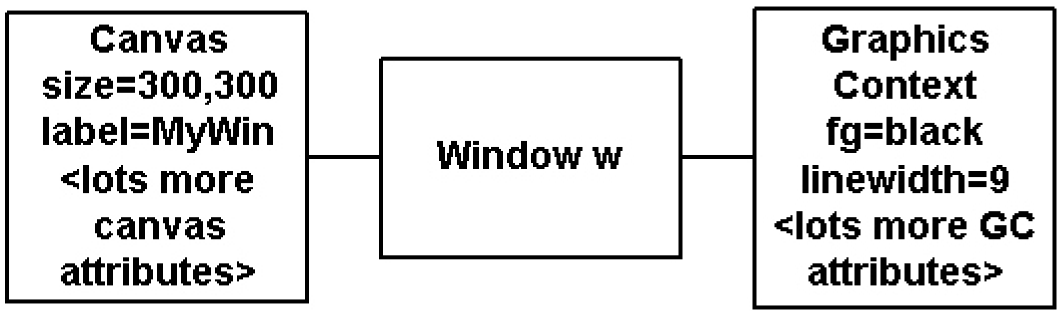
See Appendix G in the text for a complete list.

Windows, canvases, and GCs, continued

This statement:

```
    w := WOpen("size=300,300","label=MyWin",
               "linewidth=9")
```

Creates this coupling:



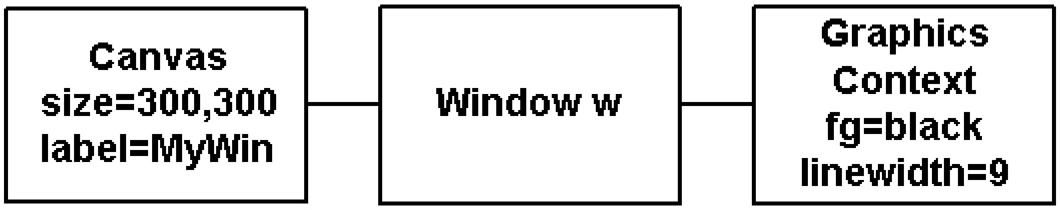
The various graphics procedures use information from the canvas and/or the graphics context to perform the desired operations.

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.

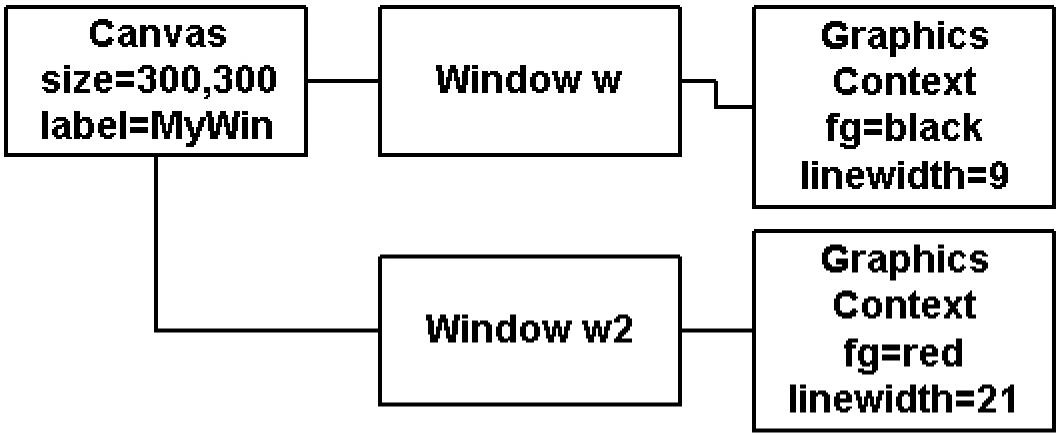
Given this coupling:



the statement

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

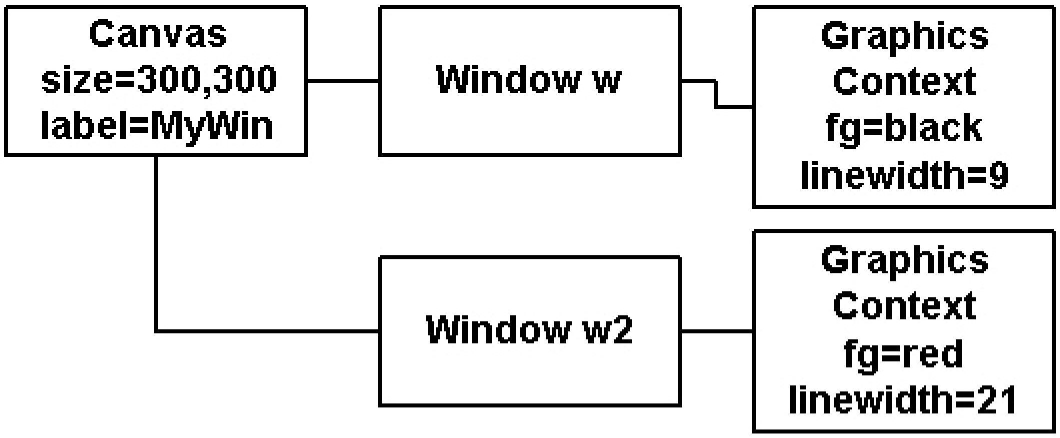
produces this:



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 `w2` will be red and 21 pixels wide.

Example:

```
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.

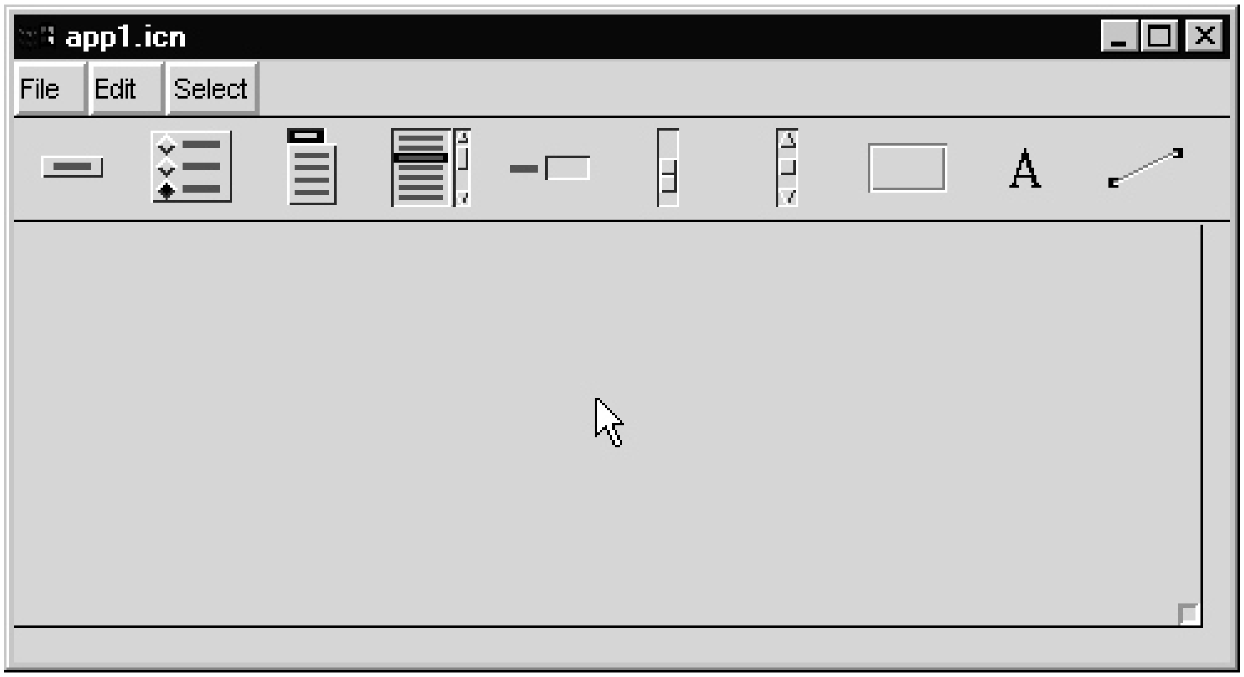
```
procedure main() # clone2
    left := WOpen("size=600,300")
    #
    # Constrain drawing to left half of window
    Clip(left, 0, 0, 300, 300)

    #
    # 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)
                }
            }
        }
end
```

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

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:



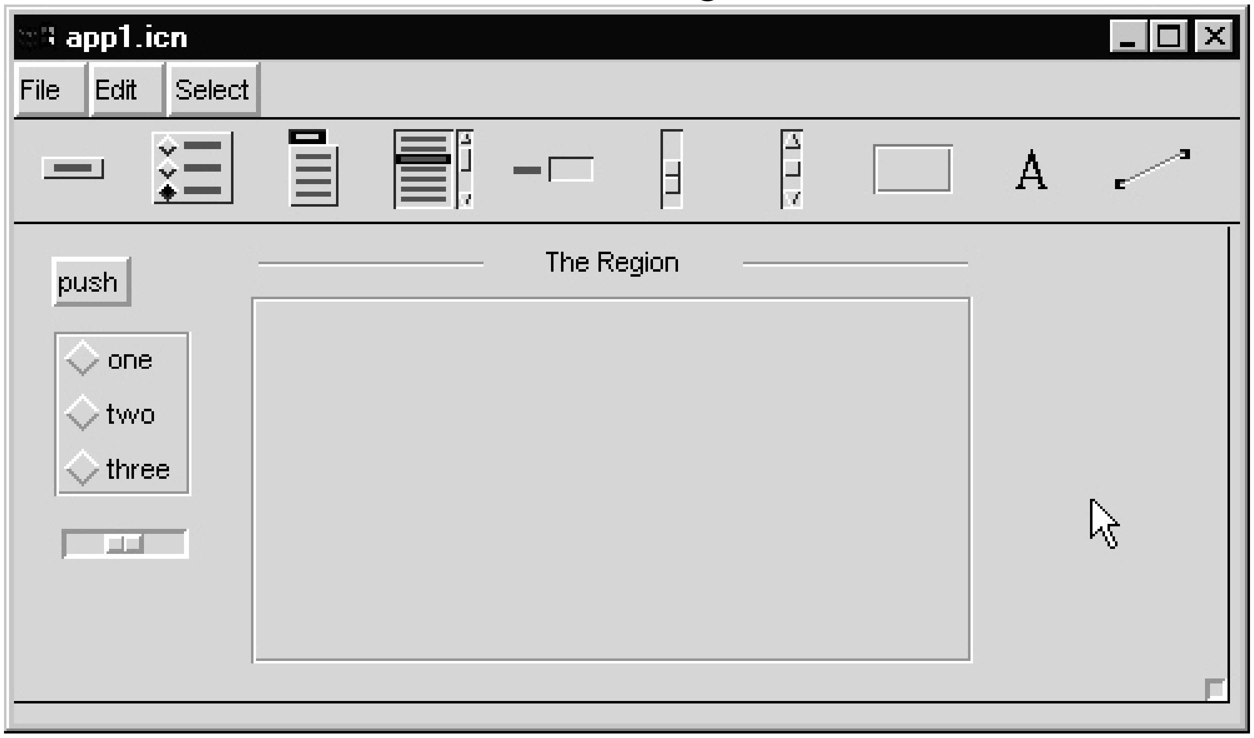
The icons below the menu bar represent the available vidgets:

Button	Scrollbar
Radio buttons	Region
Text list	Label
Text entry	Line
Slider	

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.

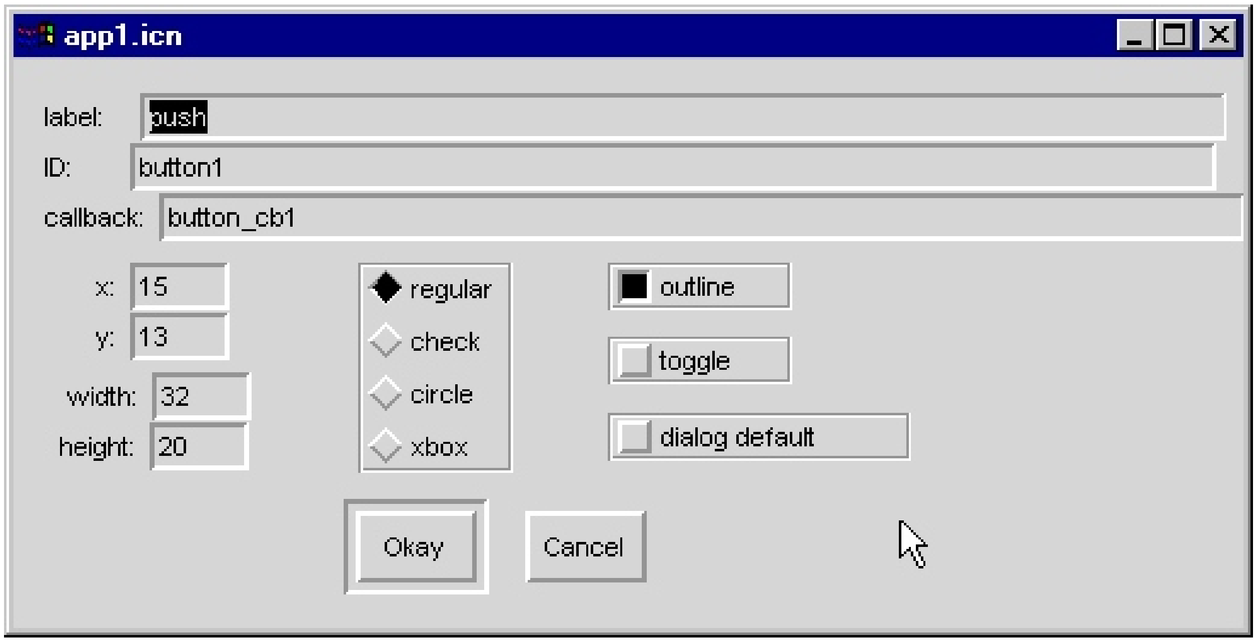
Here is an interface with several vidgets:



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:



app1.icn

label: push

ID: button1

callback: button_cb1

x: 15

y: 13

width: 32

height: 20

[✓ regular] check circle xbox

[✓ outline] toggle dialog default

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.

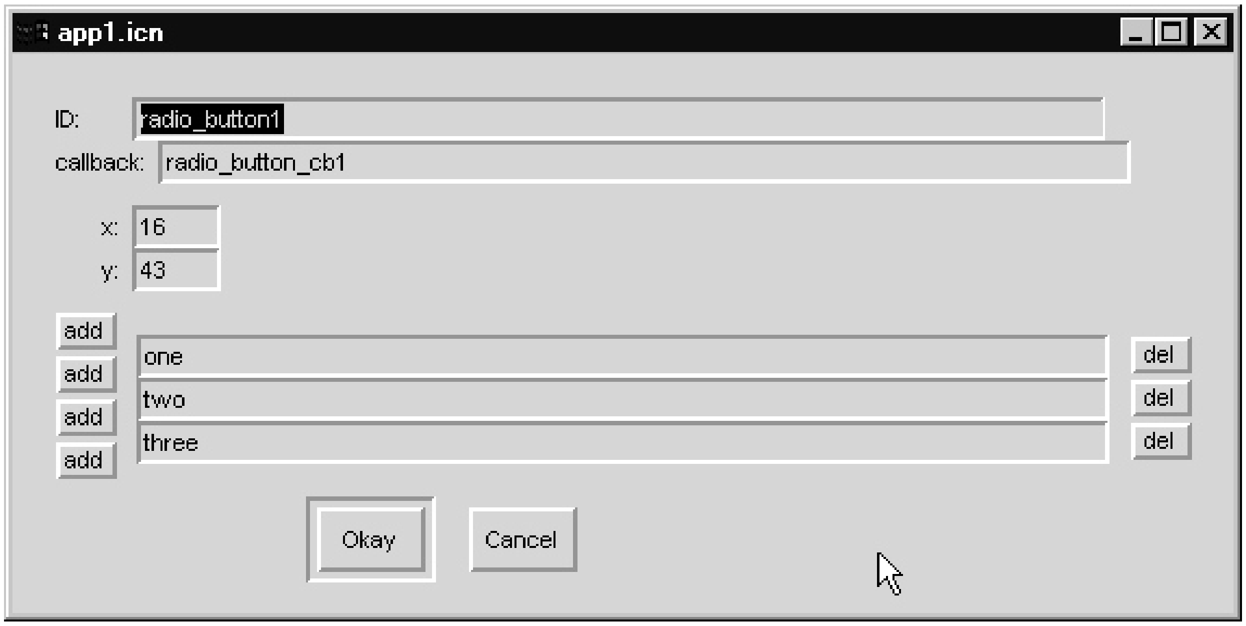
`regular`, `check`, etc. and `outline` specify details of the button's appearance.

`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:


app1.icn

ID: radio_button1

callback: radio_button_cb1

x: 16

y: 43

one

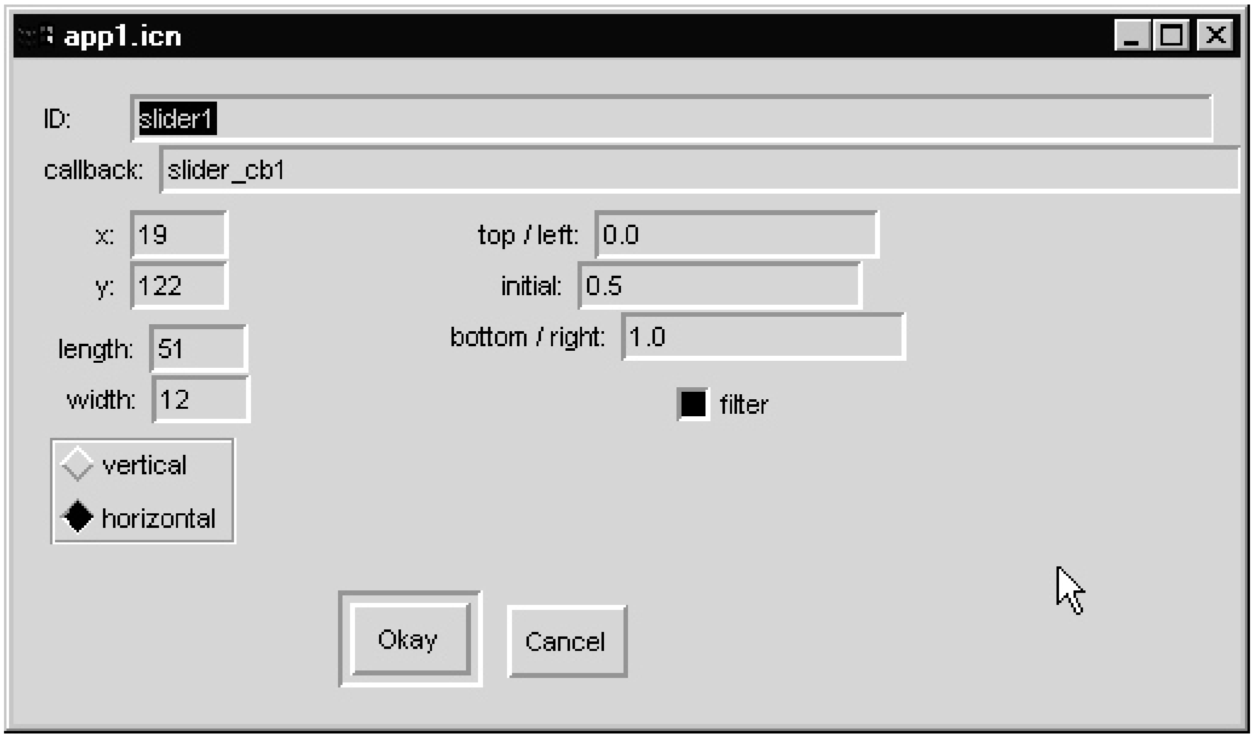
two

three

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:



ID: slider1
callback: slider_cb1
x: 19
y: 122
length: 51
width: 12
◇ vertical
◆ horizontal
top / left: 0.0
initial: 0.5
bottom / right: 1.0
■ filter

`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 `app1.icn` if no file by that name exists. If `app1.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 <u>no such check is made if the windowing system exit is actuated</u>.

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 vidgets that have been placed. Use it to select a vidget that is obscured.

Details on using VIB, continued

The `Edit|align vert` operation is used to vertically align vidgets. To use it:

(1) Select a vidget.

(2) Click on `Edit|align vert`.

(3) Clicking on a vidget to cause its Y-coordinate to be set to match the vidget selected in the first step.

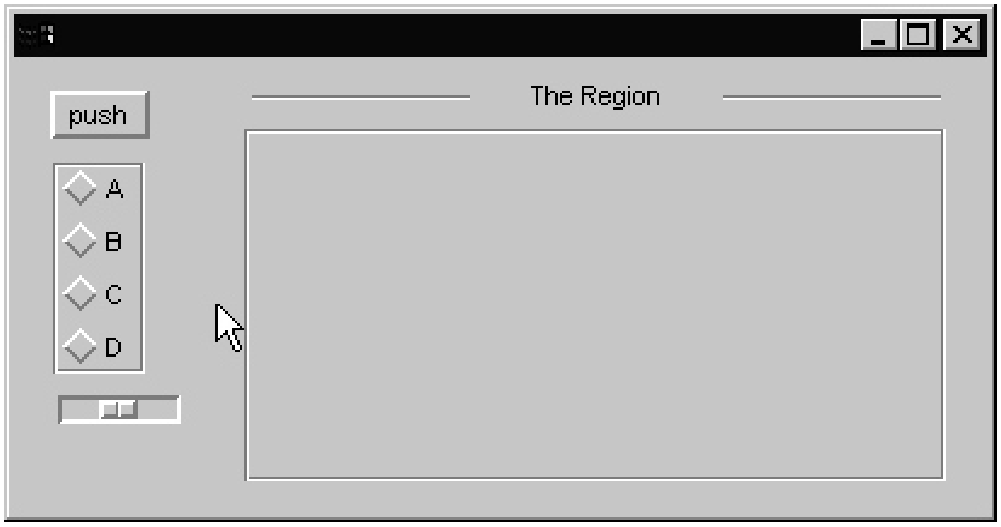
(4) When all vertical adjustments have been made, click on the canvas (i.e., not on a vidget) to exit the alignment mode.

The operation of `Edit|align horz` operation is similar, but for horizonal 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:

```
#===<<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 vidget.

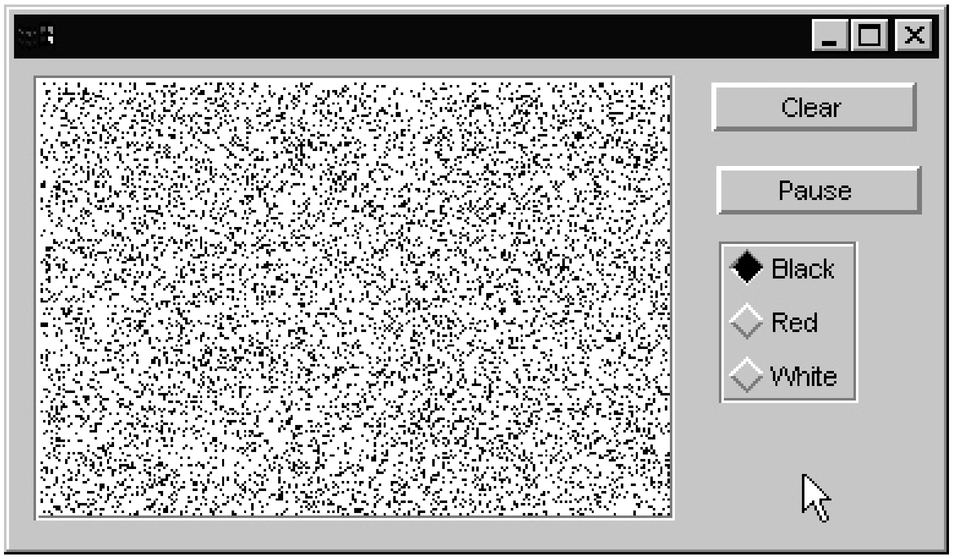
NOTE: The main routine and the callbacks are generated only on VIB's initial run for the application. On subsequent runs VIB only manipulates the `ui_atts()` and `ui()` routines.

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:

```
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:

```
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:

```
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`:

```
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 `paused`:

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

Because the Pause button is declared as a toggle, `value` will be `1` when the button is toggled on, and `&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:

```
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:

```
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 `draw_points`:

```
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
```

Finally, here is a callback for the Clear button:

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

Because the Clear button is a rebounding button, `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:

```
procedure region_cb1(vidget, 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`.

```
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

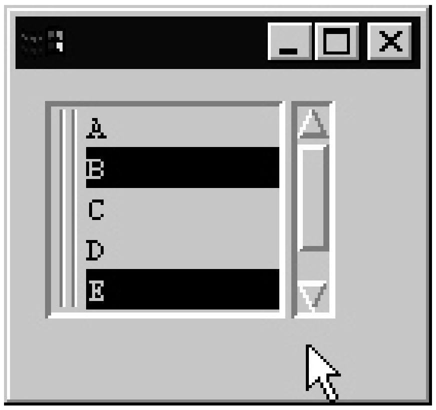
```
procedure draw_points(W)
    static width, height
    initial {
        width := WAttrib(W, "width")
        height := WAttrib(W, "height")
        }

    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
```

Text lists

The text list vidget displays a scrollable list of lines of text. Here is a text list with the letters A-F:



A text list can be configured as "select one", "select many", or "read only". The list can be scrolled vertically but not horizontally.

The *items* of a text list can be set with `VSetItems()`:

```
    VSetItems(vidgets["list1"],
              ["A","B","C","D","E","F"])
```

There is no provision for adjusting the list other than to call `VSetItems()` with a different list of values.

The list can be retrieved with `VGetItems(`*V*`)`.

Text lists, continued

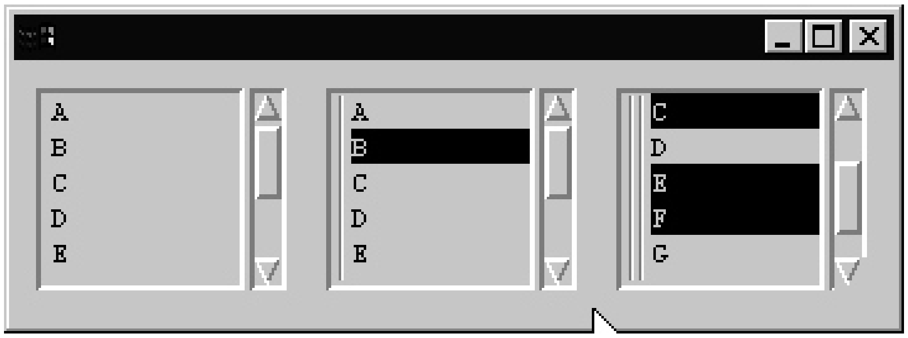
Here is the VIB-generated callback for a text list:

```
procedure list_cb1(vidget, value)
   return
end
```

If the list is single-selection, clicking on an item (e.g., "A") produces a callback with `value` equal to `"A"`.

If the list is multiple-selection, the callback is invoked with a list of the currently selected items, such as `["A"]`, `["B","E"]`, or `[]` (if no items are selected).

The *state* of a list can be retrieved with `VGetState(V)`. The value produced is a list. The first element is the index of the first visible list entry. The following elements are the indices of the selected entries, if any.

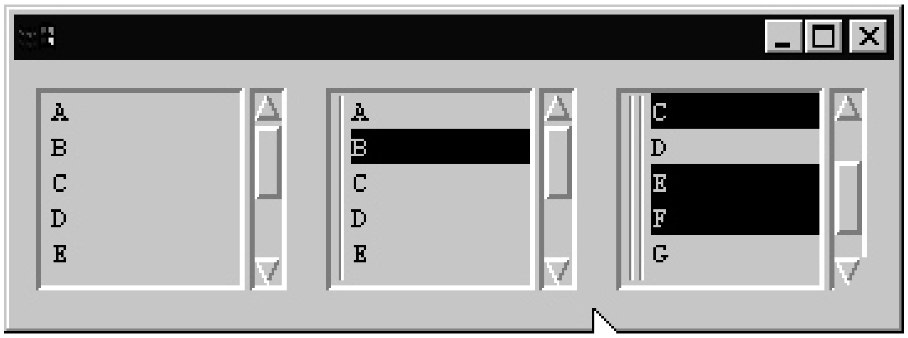


(`textlist2`)

```
Vidget: list2
Value: "B"
State: [1,2]

Vidget: list3
Value: ["C","E","F"]
State: L7:[3,3,5,6]
```

Text lists, continued

For reference:



Here is the pertinent code:

File scope:

```
    global vidgets
```

In `main`:

```
    every VSetItems(vidgets["list"||(1 to 3)],
                    ["A","B","C","D","E","F","G","H"])
```

In `list_cb`:

```
    procedure list_cb(vidget, value)
        vidget := vidgets[vidget.id] # REQUIRED!?!
        write("Vidget: ", vidget.id)
        write("Value: ", Image(value,3))
        write("State: ", Image(VGetState(vidget),3))
        write()

        return
    end
```

Note that the same callback, `list_cb`, is specified for all three lists.

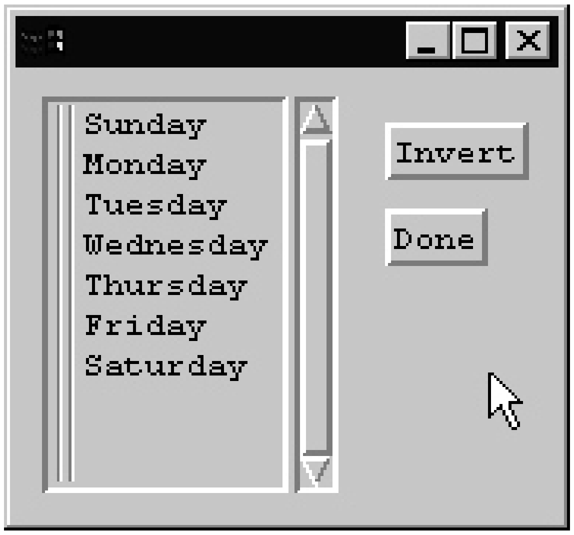
`sel`—A line selection tool

`sel` reads lines on standard input and displays a text list containing the lines. The user then indicates which lines are of interest and then `sel` prints them on standard output.

If the file `days` contains the days of the week, the command

```
    sel < days
```

displays this:



Clicking on "Sunday", then "Friday", then `Done` would produce two lines of output. The same clicks, then `Invert`, then `Done`, would produce five lines of output.

This tool might be used to produce an argument list for another command:

```
    % rm `ls | sel`
    % tar cvf x.tar `ls | sel`
```

(The shell construct `` `x` `` runs the command `x` and substitutes the output in the command line.)

`sel`—Implementation

In `main`, just above the event-processing loop:

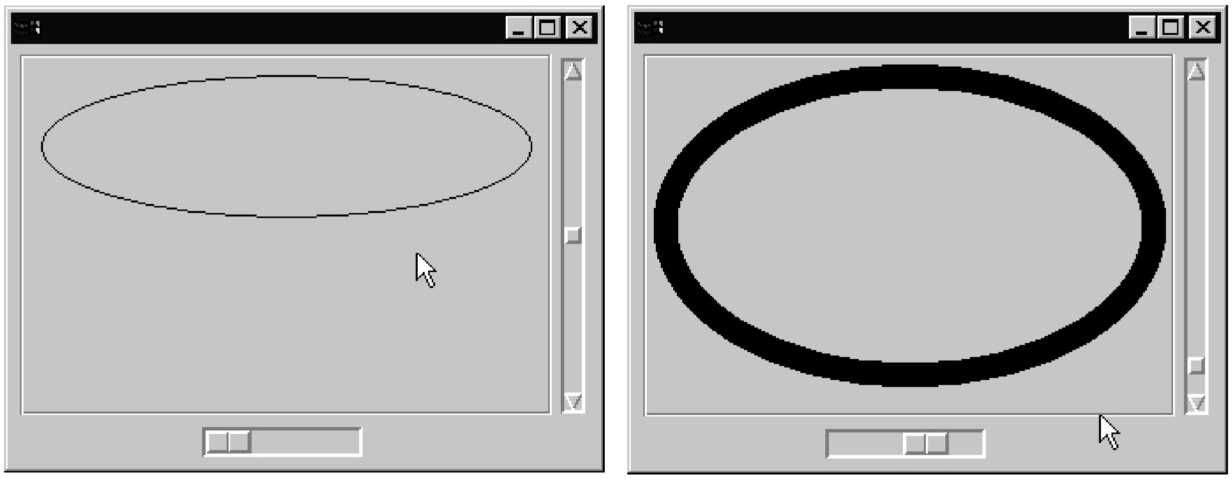
```
every put(items := [], !&input)
VSetItems(vidgets["list1"], items)
```

Callbacks:

```
global selected
procedure list_cb1(vidget, value)
   selected := value
   return
end

#
# To invert the list we first query the state and
# then build a list, 'inverted', that contains
# every position that doesn't appear in the current
# state.
#
# Example: With a five item list, if the state is
# [2,4] then inverted  will be [1,3,5].
#
procedure invert_cb(vidget, value)
    vidget := vidgets["list1"]  # REQUIRED!?!
    selected := VGetState(vidget)

    inverted := [get(selected)] # preserve position

    every i := 1 to *VGetItems(vidget) do
        if not (i = !selected) then put(inverted,i)

    VSetState(vidget, inverted) # calls list_cb1
    return
end

procedure done_cb(vidget, value)
   every write(!selected)
   exit()
end
```

Sliders and scrollbars

Consider a program, `adjust`, that permits adjustment of the "height" and line width of an ellipse via a scrollbar and a slider:



The implementation is simple: Redraw the ellipse whenever the slider or scrollbar is adjusted, using the current state of the two vidgets to control the height and line width.

The slider is configured with a minimum value of 1 and a maximum of 20, directly specifying a line width. The initial value is 1. Filtering is turned off.

The scrollbar uses the default range of 0.0 to 1.0 and an initial value of 0.5. Filtering is turned off.

Sliders and scrollbars, continued

As in the `rpoints` example, a cloned window (`ewin`) that corresponds to the region is established:

```
procedure setup_ewin()
    r := vidgets["region1"]
    ewin := Clone(&window,
        "dx="||r.ux, "dy="||r.uy)
    Clip(ewin, 0, 0, r.uw, r.uh)
end
```

Note that both `ewin` and `vidgets` are declared as globals.

Upon an adjustment we simply call `draw()`, which actually draws the ellipse. The same callback can be used by both the slider and the scrollbar:

```
procedure adjust_cb(vidget, value)
    draw()
    return
end
```

In `main`, we simply create `ewin` and call `draw()` to get the initial ellipse:

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

setup_ewin()    # Added
draw()          # Added

paused := 1
repeat {
...
```

Sliders and scrollbars, continued

Here is the drawing routine:

```
procedure draw()
    static height, width
    initial {
        width := WAttrib(ewin, "clipw")
        height := WAttrib(ewin, "cliph")
        }

    EraseArea(ewin)

    curheight :=
        VGetState(vidgets["sbar1"]) * (height-20)

    WAttrib(ewin, "linewidth=" ||
        VGetState(vidgets["slider1"]))

    DrawArc(ewin, 10, 10, width-20, curheight)
end
```

`VGetState()` is used to query the positions of both the scrollbar and the slider.

The range of the scrollbar (`sbar1`) is 0.0 to 1.0 and that value is scaled by the height.

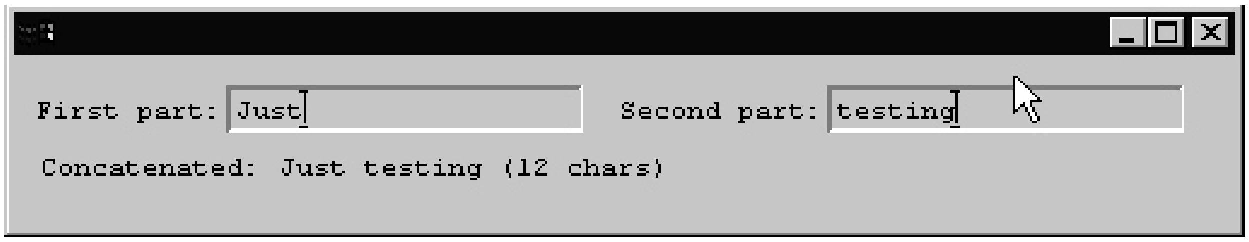
The range of the slider is 1 to 20 and that value is used as the line width.

An alternative to the `VGetState()` calls would be to use separate callbacks for the slider and scrollbar. The slider callback would use `WAttrib()` to set the line width. The scrollbar callback would put the value passed to the callback (the second argument) in a global variable that would be accessed in `draw()`.

The 10s and 20s simply provide centering of the ellipse.

Text vidgets

A text vidget consists of a label and a field in which to type text. This application, `text1`, has two text vidgets and, on the line below, a label. To the right of the label is a region with an invisible border.



```
First part: Just          Second part: testing
Concatenated: Just testing (12 chars)
```

The concatenation of the text entered in the two text vidgets, and the length, is displayed.

Text vidgets are somewhat limited in functionality. Characters are recognized only when the mouse is over the vidget. A callback is generated only when the user presses return and until the user the presses return, `VGetState()` returns null.

Further, due to a bug in the Windows implementation **the input-sensitive region is not aligned what's drawn on the screen**. The atrocious but currently best workaround for this is to position the window so that the upper left corner of the canvas is in the upper left corner of the display. On Windows NT, this does it:

```
    WAttrib("pos=-4,-23")
```

Text vidgets, continued

The callback for each text vidget simply stores the value in a global variable, and calls a routine to update the result:

```
    procedure text_input_cb1(vidget, value)
        first_part := value
        show_result()
        return
    end
```

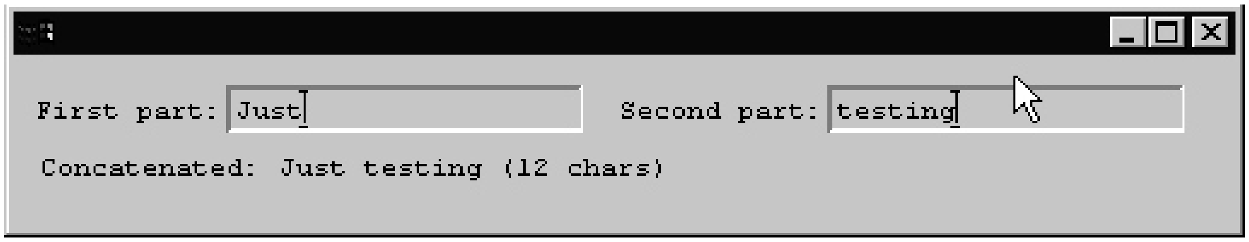
`text_input_cb2` is similar, but assigns to `second_part`.

Text vidgets, continued

The common way to produce computed text on a VIB interface is to create an invisible region where the text is to appear and use the coordinates and size of that region to control the output.

Here is a utility routine that is like `WWrites()`, but accepts a region vidget (or its ID) as its first argument and writes the text of the following arguments into the region, truncating appropriately.

```
procedure RWWrites(rvidget, args[])
    r := \vidgets[rvidget] | rvidget
    GotoXY(r.ux, r.uy+WAttrib("ascent"))

    s := ""
    every s ||:= !args
    WWrites(left(s, r.uw/TextWidth("x")))
    return
end
```



```
First part: Just          Second part: testing

Concatenated: Just testing (12 chars)
```

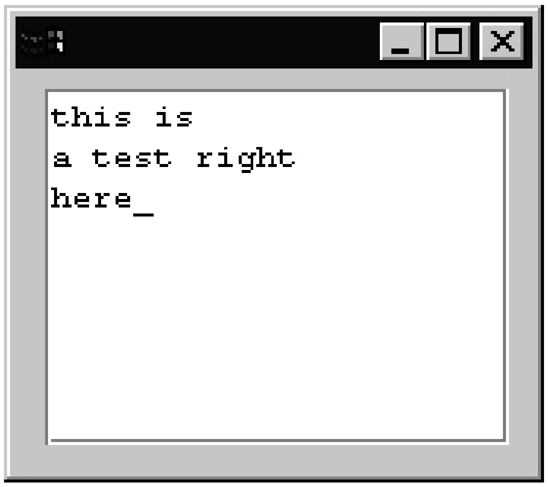
Use `link whmvib` to access the routine.

Here is the routine called by the text vidget callbacks:

```
procedure show_result()
    RWWrites("region1",
        s := first_part || " " || second_part,
        " (", *s, " chars)")
end
```

Example: A text window

Here is an example of using keyboard events from a region to implement a very simple text editing window.

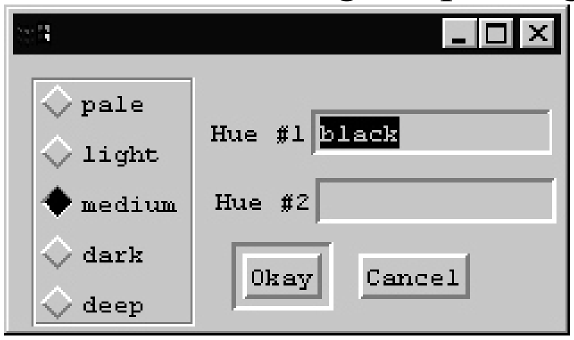


```
this is
a test right
here_
```

```
procedure region_cb1(vidget, e, x, y)
    static r
    initial r := ""
    case e of {
        default: if type(e) == "string" then {
            case e of {
                "\r": r ||:= "\r\n"
                "\b": if r[-1] == "\n" then
                          r[0-:2] := ""
                      else
                          r[-1] :=  ""
                default:
                    r ||:= e
                }
            EraseArea(point_win)
            GotoRC(point_win,1,1)
            WWrites(point_win,r,"_")
            }
        }
   return
end
```

VIB dialogs

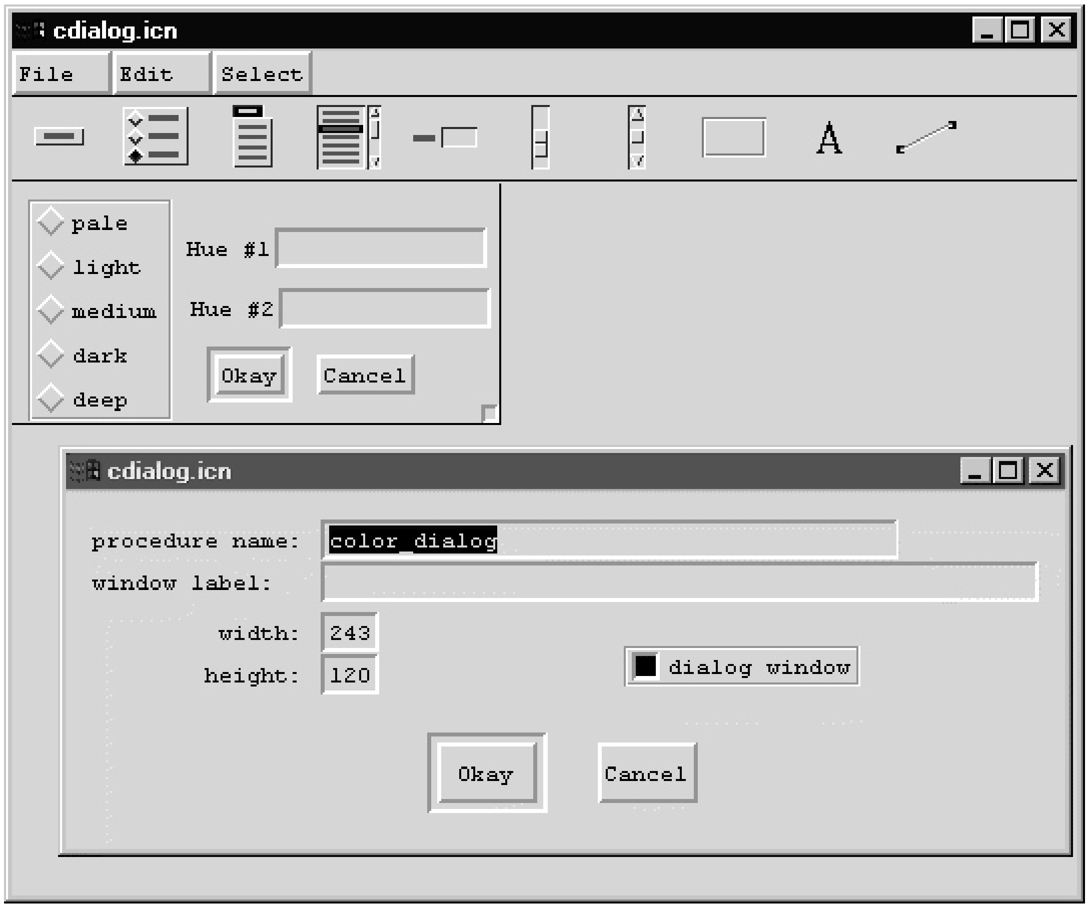
In addition to creating program interfaces, VIB can create dialog boxes. Here is a dialog for picking a color:



Hue #1 black

Hue #2

Use VIB as usual to create the interface and then click on the lower right corner to bring up the interface properties. Set the procedure name (`color_dialog`) and select "dialog window".



procedure name: color_dialog

window label:

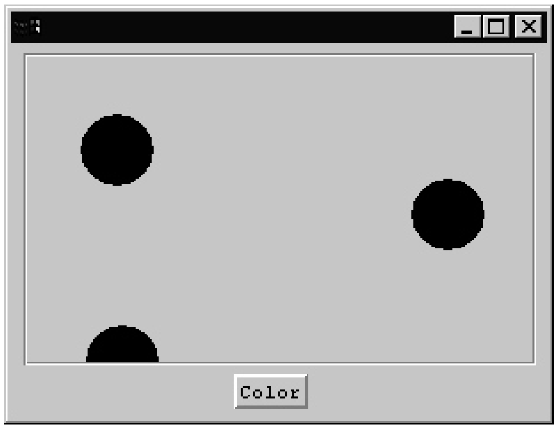
width: 243

height: 120

[✓ dialog window]

VIB dialogs, continued

Here is a simple program to exercise the dialog: (`dlg1`)



By default, a black circle is drawn in response to a click in the region. Pressing the `Color` button brings up the color selection dialog to specify the color of subsequent circles.

VIB dialogs, continued

The color dialog is handled entirely in the callback for the `Color` button:

```
procedure color_cb(vidget, value)
    attribs := table()
    attribs["lightness"] := "medium"
    attribs["hue1"] := "black"

    if color_dialog(&window, attribs) == "Okay"
    then
        Fg(dwin, attribs["lightness"] || " " ||
            attribs["hue1"] || " " ||
            attribs["hue2"])
    return
end
```

The dialog is brought up by calling `color_dialog`, the procedure name specified in the interface properties, as shown on slide 95.

Two arguments are passed to `color_dialog`: the window to associate the dialog with, and a table specifying initial values for the vidgets in the dialog.

When the dialog is dismissed, `color_dialog` returns the label of the button used to dismiss it.

In the above example, if the user pressed the `Okay` button, the color for `dwin` (a cloned window for the region) is set based on values in `attribs`, which in turn were set to the vidget values present when the dialog was dismissed.

VIB dialogs, continued

There can only be one VIB-generated interface in a source file, so each dialog must be in its own source file.

The color dialog was created with '`vib cdialog.icn`'. Here is `cdialog.icn`, minus comments:

```
link dsetup
#===<<vib:begin>>=== modify using vib;
procedure color_dialog(win, deftbl)
static dstate
initial dstate := dsetup(win,
   ["color_dialog:Sizer::1:0,0,243,120:",],

["button1:Button:regular:-1:101,86,35,20:Okay",],

["button2:Button:regular::152,86,49,20:Cancel",],
   ["hue1:Text::14:87,23,150,20:Hue #1\\=",],
   ["hue2:Text::14:89,53,150,20:Hue #2\\=",],
   ["lightness:Choice::5:8,9,72,110:",,
      ["pale","light","medium","dark","deep"]],
   )
return dpopup(win, deftbl, dstate)
end
#===<<vib:end>>===
```

The program was built with the command

```
icont dlg1.icn cdialog.icn
```

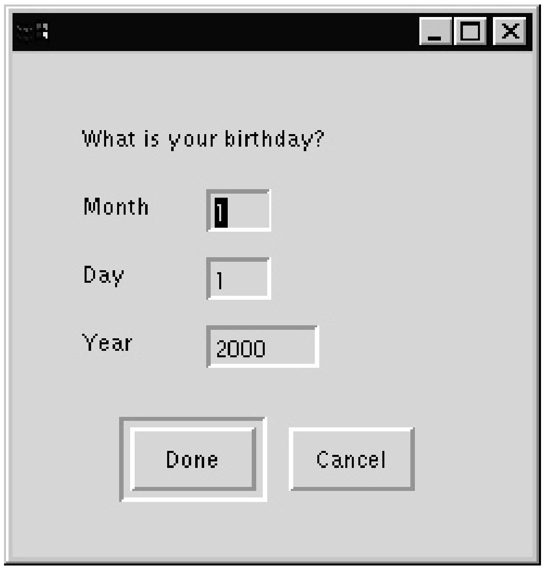
Note that VIB assumes that non-toggling buttons dismiss the dialog. If a button is specified (via VIB) as "dialog default", then hitting <Enter> will simulate a button press on it.

Utility dialogs

There are several other utility dialogs, such as `Notice()`. One is `TextDialog`, which presents a set of labeled text vidgets. This program:

```
global dialog_value
procedure main() # dlg2
    rslt := TextDialog("What is your birthday?",
        ["Month", "Day", "Year"],
        [1,1,2000],
        [2,2,4],  # field widths
        ["Done","Cancel"])

    write(Image([rslt, dialog_value],3))
end
```

Produces this dialog:


What is your birthday?

Month 1

Day 1

Year 2000

When dismissed, the global `dialog_value` is set to the values of the fields and the label of the button pressed is returned.

The above `Image` might produce this:

```
    ["Done",["7","4","1976"]]
```

Utility dialogs, continued

Other utility dialogs are:

`SelectDialog`—displays a set of radio buttons

`ToggleDialog`—displays a set of toggle buttons

`ColorDialog`—allows selection of a color using the RGB or HSV color models

`OpenDialog`, `SaveDialog`—simple front-ends to `TextDialog` to prompt for file names

Chapter 14 in the graphics text describes the utility dialogs in detail.