```
  end

  ##########
  var got, tmp : int
  do true ->
    in ec(ev) ->
      if (ev.event_type = Ev_KeyUp) ->

        var ch := char(ev.data)
        if ch = 'q' | ch = 'Q' | ch = '\177' | ch = '\003' ->
          stop
        [] ch = ' ' ->
          if not started ->
            started := true
            WinClearArea(mainwin, winRectangle(0, 0, SIZE, SIZE+DOT))
            banner(textw, MSG_RUNNING)
            got := 0
            send job(1, NP)
          [] else ->
            WinBell(mainwin, 0);
          fi
        [] ch = '-' ->
          slp +:= 5
        [] ch = '+' ->
          slp := max(slp - 5, 0)
        [] ch = 'r' ->
          if not started ->
            regenerate()
            banner(textw, MSG_READY)
          [] else ->
            WinBell(mainwin, 0)
          fi
        fi

      [] ev.event_type = Ev_DeleteWindow ->
        stop
      fi

    [] done(tmp) ->               # in
      got +:= tmp
      if got >= NP ->
        banner(textw, MSG_DONE)
        started := false
      fi
    ni
  od

  ### Final
  final
    WinClose(mainwin)
  end

end qsort
```

```
######################################################################
# open main window
mainwin := WinOpen("", "Quick Sort", ec, UseDefault, SIZE, SIZE+OFF)
if mainwin = null ->
  write("Ouch, can't open window")
  stop(1)
fi
WinSetEventMask(mainwin, Ev_KeyUp)
ww := WinNewContext(mainwin)
boxw := WinNewContext(mainwin)
textw := WinNewContext(mainwin)
fa i:= 1 to JS ->
  cwins[i] := WinNewContext(mainwin)
  WinSetForeground(cwins[i], colors[i])
af
WinSetForeground(mainwin, "black")
WinSetForeground(boxw, "orange")
WinSetForeground(textw, "lightyellow")
WinSetLineAttr(boxw, 0, LineDoubleDash, CapButt, JoinMiter)

## a taste of flavor
seed(0)

## sorting control
regenerate()
banner(textw, MSG_READY)
slp := 40

## workers, get job from bag...
process worker(id := 1 to JS)
  var lo, mid, hi: int
  var x, y, w, h: int
  var mywork: int := 0

  receive job(lo, hi)
  do true ->
    mid := part(id, lo, hi, x, y, w, h)
    WinDrawRectangle(boxw, winRectangle(x, y, w, h))
    if lo < mid - 1 ->
      if mid + 1 < hi ->
        send job(mid+1, hi)
      [] else ->
        mywork +:= hi - mid + 1
      fi
      hi := mid -1
    [] else ->
      mywork +:= mid - lo + 1
      if mid + 1 < hi ->
        lo := mid + 1
      [] else ->
        mywork +:= hi - mid + 1
        if lo = hi -> mywork -- fi              # one work counted twice
        send done(mywork)
        mywork := 0
        receive job(lo, hi)
      fi
    fi
  od
```

```
  fa i:= 1 to NP ->
    numbers[i] := int(random(SIZE)+1)
    drawDot(ww, i, numbers[i], 0)
  af
end

## Draw a box stands for a point being sorted
proc drawDot(win, idx, value, slp)
  if slp > 0 -> nap(slp) fi
  WinDrawRectangle(win,
    winRectangle(int(SCALE*(idx-1)), SIZE-value-DOT2, DOT, DOT))
end

###
# Partition
proc part(id, left, right, x, y, w, h) returns mid
  var minv := SIZE+1, maxv := -1
  var i : int
  fa i := left to right ->
    minv := min(minv, numbers[i])
    maxv := max(maxv, numbers[i])
  af
  x := int((left-1) * SCALE + DOT2)
  h := maxv - minv + 1
  y := SIZE - minv - h
  w := int((right-left) * SCALE)
  #
  WinDrawRectangle(boxw, winRectangle(x, y, w, h))
  # recolor to my color
  fa i:= left to right ->
    drawDot(cwins[id], i, numbers[i], 0)
  af
  WinSync(mainwin, false)             # intended to slow things down
  #
  var pivot := numbers[left]
  var lx := left+1, rx := right
  do lx <= rx ->
    if numbers[lx] <= pivot -> lx++
    [] numbers[lx] > pivot ->
      drawDot(mainwin, lx, numbers[lx], 0)
      drawDot(mainwin, rx, numbers[rx], 0)
      numbers[lx] :=: numbers[rx]
      drawDot(cwins[id], lx, numbers[lx], slp)
      drawDot(cwins[id], rx, numbers[rx], slp)
      WinSync(mainwin, false)         # intended to slow things down
      rx--
    fi
  od
  if rx > left ->
    drawDot(mainwin, left, numbers[left], 0)
    drawDot(mainwin, rx, numbers[rx], 0)
    numbers[rx] :=: numbers[left]
    drawDot(cwins[id], left, numbers[left], slp)
    drawDot(cwins[id], rx, numbers[rx], slp)
  fi
  #
  mid := rx
  return
end
```

# Appendix C. Sample Program - Quick Sort

```
# Quick Sort
#
# Qiang A. Zhao, October 92
#
# Usage: qsort WindowSize Points DotSize

resource qsort()

  const JS := 4                         # number of job servers
  import SRWin

  op banner(win: winWindow; str: string[*])
  op regenerate()
  op drawDot(win: winWindow; idx, value, slp: int)
  op part(id, left, right: int; var x, y, w, h: int) returns mid:int
  op job(left, right: int)
  op done(int)

  var SIZE := 500;      getarg(1, SIZE)
  var NP := SIZE/2;     getarg(2, NP)
  var DOT := 6;         getarg(3, DOT)

  if (SIZE < 100) or (NP < 10) ->
    write("Invalid Value, sorry...")
    stop(1)
  fi

  const OFF := 40
  const DOT2 : int := DOT/2
  const SCALE : real := real(SIZE-DOT)/real(NP-1)

  const MSG_READY := "Press 'SPACE' to go"
  const MSG_RUNNING := "Running: '+'/'-' to adjust speed"
  const MSG_DONE := "Done, 'r' to regenerate numbers"

  var numbers[NP] : int
  var mainwin, ww, boxw, textw: winWindow
  var cwins[1:JS] :winWindow
  var colors[1:JS] : string[6]
        colors[1] := "red"
        colors[2] := "yellow"
        colors[3] := "green"
        colors[4] := "cyan"
  op  ec : winEventChannel
  var ev: winEvent
  var slp: int := 0
  var started := false

  ## Draw a string at the bottom
  proc banner(win, str)
    WinClearArea(mainwin, winRectangle(0, SIZE+DOT, SIZE, OFF-DOT))
    WinDrawString(win, winPoint(OFF, SIZE+30), str)
  end

  ## Generate random numbers
  proc regenerate()
    WinClearArea(mainwin, winRectangle(0, 0, SIZE, SIZE+DOT))
```

```
    var pressed : bool := false
    do true ->
      var ev: winEvent
      receive evchan(ev)

      if ev.event_type = Ev_ButtonDown ->
        pressed := true
        WinSetBorder(mywin, but.border, borderColor)
        WinEraseArea(revw, winRectangle(0, 0, w, h))
        WinDrawString(revw, winPoint(labx, laby), labelString)

      [] ev.event_type = Ev_ButtonUp ->
        WinSetBorder(mywin, but.border, borderColor)
        WinEraseArea(normalw, winRectangle(0, 0, w, h))
        WinDrawString(normalw, winPoint(labx, laby), labelString)
        if pressed ->
          cb(but, labelString)
        fi
        pressed := false

      [] ev.event_type = Ev_EnterWindow ->
        WinSetBorder(mywin, but.border, borderColor)

      [] ev.event_type = Ev_ExitWindow ->
        pressed := false
        WinSetBorder(normalw, but.border, buttonColor)
        WinEraseArea(normalw, winRectangle(0, 0, w, h))
        WinDrawString(normalw, winPoint(labx, laby), labelString)
      fi

      WinFlush(mywin)
    od
  end
end Button      ### global

resource ButtonTest()
  import SRWin, Button

  op cb1, cb2: buttonCallBack

  proc cb1(b, str)
    write("Hello")
  end

  proc cb2(b, str)
    write("Quit"); stop(0)
  end

  var mywin: winWindow := WinOpen("", "Button Test", null, UseDefault, 130, 80)
  WinFlush(mywin)
  send button(buttonRec(mywin, 10, 10, 50, 40, 2),
            "red", "white", "blue", "Hello", cb1)
  send button(buttonRec(mywin, 70, 10, 50, 40, 2),
            "green", "blue", "white", "Quit", cb2)

  final
    WinClose(mywin)
  end
end ButtonTest
```

# Appendix B. Sample Program - Button

```
# "button.sr"                              Q. A. Zhao, March 93

global Button

  import SRWin

  type buttonRec = rec (
                window : winWindow
                x, y, w, h, border : int
        )

  optype buttonCallBack(buttonRec; string[*])

  op button(    but: buttonRec;
                borderColor, buttonColor, labelColor : winColor;
                labelString: string[*];
                cb: cap buttonCallBack)

body Button

  import SRWin

  ### A button
  proc button(but, borderColor, buttonColor, labelColor, labelString, cb)
    var labelFont : winFont := WinDefaultFont(but.window)

    var x := but.x + but.border
    var y := but.y + but.border
    var w := but.w - but.border * 2
    var h := but.h - but.border * 2
    var labx := (w - WinTextWidth(labelFont, labelString)) / 2
    var laby := (h + WinFontAscent(labelFont) - WinFontDescent(labelFont)) / 2

    var mywin: winWindow
    op evchan: winEventChannel
    mywin := WinCreateSubwindow(but.window, evchan, OffScreen, x, y, w, h)
    if mywin = null ->
      write(stderr, "Button: '", labelString, "' cannot be created")
      return
    fi

    WinSetEventMask(mywin, Ev_ButtonDown | Ev_ButtonUp |
        Ev_EnterWindow | Ev_ExitWindow)

    var normalw : winWindow := WinNewContext(mywin)
    WinSetForeground(normalw, labelColor)
    WinSetBackground(normalw, buttonColor)

    var revw : winWindow := WinNewContext(mywin)
    WinSetForeground(revw, buttonColor)
    WinSetBackground(revw, labelColor)

    WinSetBorder(normalw, but.border, buttonColor)
    WinEraseArea(normalw, winRectangle(0, 0, w, h))
    WinDrawString(normalw, winPoint(labx, laby), labelString)
    WinMapWindow(normalw)               # same as mapping 'mywin'
```

Icon, but that have not yet been implemented. These include mutable color cells, reading and writing images from and to files, etc.

While providing more user interface features is an important development direction, tuning SR Win for better performance is also essential. One possible thread of revising SR Win is to rewrite the locking scheme after the multi-threaded Xlib is available [Gild93]. For example, based on the new Xlib, SR Win would no longer need to lock Xlib data structures and functions entries. Only data structures created and used by SR Win must be protected by locking. The semaphore could be made on a per -connection basis to further relieve processes working on one display from competing for locks with those working on other displays.

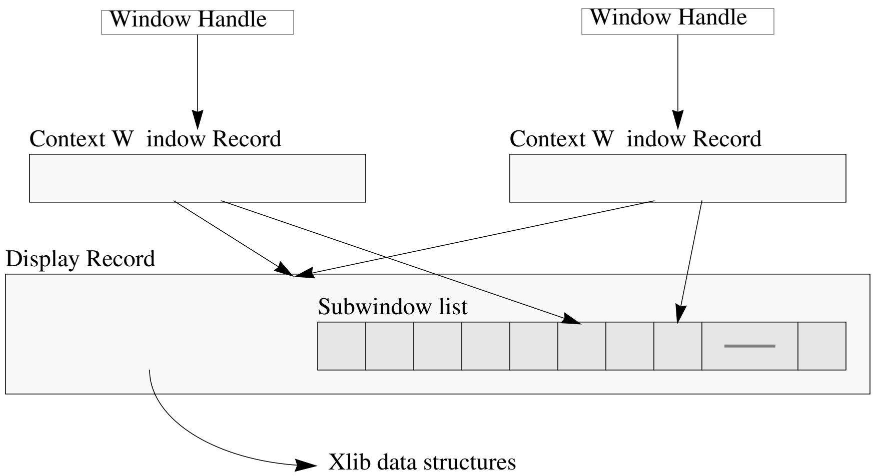


Figure 7. Objects in SR Win

Function **`WinOpen()`** creates the display record and the first context window record. **`WinCreateSubwindow()`** inserts a subwindow into the subwindow list and creates a new context window record. **`WinNewContext()`** only creates a context window record. All three functions return pointers to the context window records they create.

However , SR Win doesn' t keep track of every context window record it creates. When a function like **`WinDestroyWindow()`** is called, it walks the list of subwindows and destroys those required, but it cannot destroy other context window records not passed in the list of arguments. After destroying a window , other context window handles still referring to the old window should not be used.

## Concurrency Issues

A semaphore in SR Win serializes calls to Xlib. Whenever a function is about to change something, for example load a font or draw a line, it does a **`P`** operation on that semaphore; afterwards it does a **`V`** operation on the semaphore. The operations provided without this mutual exclusion scheme include, partly because of performance concerns, **`WinTextWidth()`**, **`WinFontAscent()`**, **`WinFontDescent()`**, **`WinGetPixel()`**, **`WinPutPixel()`**, and **`WinAddPixel()`**. The first four functions read information in the memory space of the application program. They are safe as long as the subject of the operation is valid. When using the last two functions, the application program must ensure that there is no simultaneous access to the same row in the image.

When **`WinOpen()`** is called and succeeds in creating the window , it does a **`reply`** so it gets detached and becomes a separate process, the event handler . This process periodically checks whether there is an event pending from the X server . If there is any , it retrieves the event from Xlib, and calls a C function to find out which subwindow the event should be in. Then the event is dispatched to the registered event channel, if there is one. The delay between two checking iterations of the event handler is controlled by **`WinSetPoll()`**; the minimum delay value is set to 1 millisecond.

**`WinClose()`** destroys the windows and closes the connection to the X server . It resets a flag field in the display record so the event handler , which also periodically checks this field, can exit.

## Future Development

SR Win has borrowed a lot of ideas from X-Icon [Jeff92] on providing user interface primitives. At the time this document was written, there are still a lot of nice features that SR Win can learn from X-

# Appendix A. The Implementation of SR Win

A basic knowledge of the X Window System is very helpful in understanding the implementation of SR Win. Various books and articles listed in the Reference section describe the X Window System and Xlib.

## The Client-Server Model

Like many other X-based systems, SR Win follows and benefits from the client-server computing model. Conceptually, what **`WinOpen()`** establishes is a full-duplex connection between the X server and the client program. This allows outputs flow from the client to the server, and inputs flow from the server to the client:

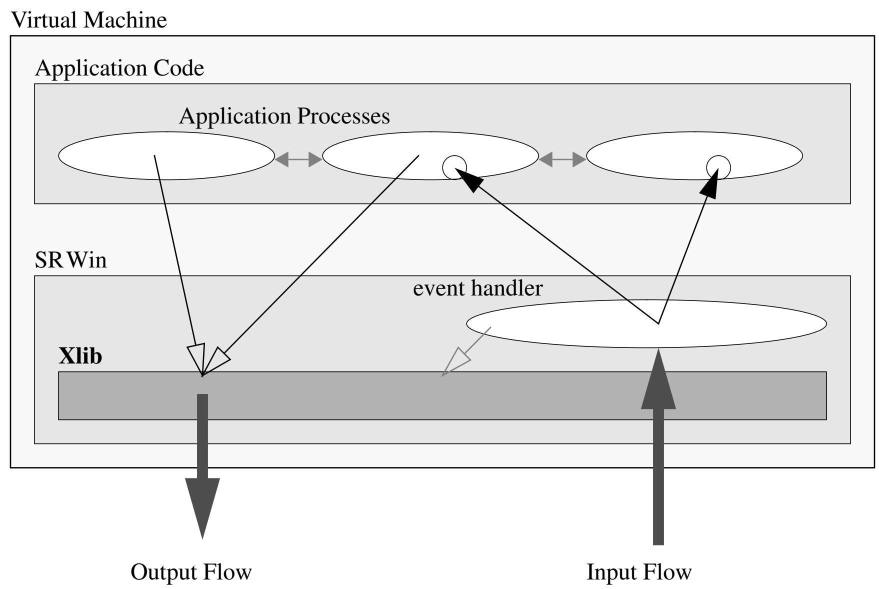


Figure 6. SR Win Internal Layout

In a *virtual machine* running a client program, SR Win acts as a global embedded with Xlib. Application processes, which may be in one or more *resources*, communicate with each other to accomplish some computation tasks. They call SR Win primitives in order to display some data or manipulate the windows. These calls are serialized by SR Win to prevent them from simultaneous access to shared critical data structures in Xlib, and they are turned into X requests by Xlib and sent to the X server. Inputs from various windows, on the other hand, are delivered to the event handler. The event handler responds to the exposure events by sending back redrawing information to the server, and forwards other events to the appropriate event channel (denoted by a little circle in the above figure) by asynchronous message passing.

## Manipulating Window Objects

The implementation of SR Win includes two parts: an SR global **`SRWin`** and a set of C functions. The C functions are called as *externals* from the SR global. There are a few type of data objects that hold state information of context windows and *displays* (connections to X server). Both the SR part and C part can access these objects (see Figure 7).

## References

[AnOl93] Andrews, G. R., and Olsson, R. A. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1993.

[Brow91] Brown, M. H., editor. *An Anthology of Algorithm Animations using Zeus.* TR 76b (videotape). Digital Equipment Corporation, Systems Research Center, Palo Alto, California, August 1991.

[Gild93] Gildea, S. Multi-Threaded Xlib. *The X Resource, Issue Five — Proceedings of the 7th Annual X Technical Conference*, pages 159-166. O'Reilly & Associates, Inc., Sebastopol, California, 1993.

[Huds93] Hudson, S. *Winpack — Minimal Retargetable Window Package*. GVU Center, College of Computing, Georgia Institute of Technology, 1993.

[Jeff92] Jeffrey, C. L. *X-Icon: An Icon Window Interface, Version 2*. TR 92-26. Dept. of Computer Science, The University of Arizona.

[OlAn92] Olsson, R. A., Andrews, G. R., Coffin, M. H., and Townsend, G. M. *SR, A Language for Parallel and Distributed Programming*. TR 92-09, Dept. of Computer Science, The University of Arizona, March 1992.

[NyeA92] Nye, A., editor. *Xlib Programming Manual, for Version 11 of the X Window System; 3rd ed*. O'Reilly & Associates, Inc., Sebastopol, California, 1992.

[ScGe86] Scheifler, R. W. and Gettys, J. The X Window System. *ACM Transactions on Graphics, Vol. 5, No. 2, April 1986*, pages 79-109.

[ScGe92] Scheifler, R. W., and Gettys, J., with Flowers, J. and Rosenthal D. *X Window System - The Complete Reference to Xlib, X Protocol, ICCCM, XLFD; 3rd ed*. Digital Press, Burlington, Massachusetts, 1992.

[Zhao93] Zhao, Q. A. *SR Win Manual Page*. Dept. of Computer Science, The University of Arizona, May 1993.

```
var i: int
fa i := 0 to 199 ->                    # a horizontal red line
  WinPutPixel(im, winPoint(i, 50), p)
af
```

Calling **`WinPutImage()`** copies a rectangular area in the image to a window:

```
# copies the image to (20, 10) on window "mywin"
WinPutImage(mywin, im, winRectangle(0, 0, 199, 99), winPoint(20, 10))
```

Unlike windows and their backup copies, an image is kept in the memory area of the application program, not the X server. Primitives like **`WinPutPixel()`** operate locally without generating any network traffic. On a shared memory parallel machine, processes on different processors can actually manipulate a shared image concurrently, resulting in faster program execution.

# 9. Conclusions

Building on top of Xlib, SRWin is a quite simple graphics library for SR. Like Xlib, SRWin provides graphics and windowing mechanisms without enforcing any particular policy, user interface, or look-and-feel. SRWin provides a few higher-level abstractions consistent with the rest of the SR language.

Many interactive programs are *event driven*, meaning that they follow a paradigm in which an event-reading loop is the primary control pattern driving the program. If a program must be prepared to respond to user input, it cannot compute for long periods without checking for window events.

SRWin, however, supports a multi-paradigm programming model. Some of the processes in an interactive system can be totally unaware that the system as a whole is interacting with the user. Other process can be solely devoted to processing user commands. This modularity greatly reduces the burden on the programmer.

Multiple processes and event channels help make programming in SRWin easy and efficient. The message passing and operation invocation methods in SR makes those abstractions feasible. The concurrency in SR gives SRWin the power of implementing retained windows, where window redrawing is handled automatically. Given the primitives presented in SRWin, advanced user interface features can be written in SR.

This document gives many examples of using SRWin. The SRWin reference manual in the SR distribution provides a complete list of SRWin operations with syntax and semantics explanations [Zhao93]. Many programming details, such as shapes of standard cursors, can be found in various X Window System books and articles listed in the "Reference" section below.


## Acknowledgments

SRWin was inspired by X-Icon, an X interface developed by Clint Jeffery for the Icon language [Jeff92]. Many useful ideas came from the *Winpack* graphics library created by Scott Hudson [Huds93]. Gregory Andrews and Gregg Townsend provided a lot of guidance on designing, implementing, and documenting the library.

## Binding Context Information With Windows

Usually a graphics primitive does not contain all the information needed to draw a particular thing. The X server maintains resources called *graphics contexts* that specify many attributes that apply to each graphic request.

Different graphics contexts can be used to write to the same window. An SR Win window handle is actually a *binding* of a window on the screen with a particular graphics context. Function **`WinOpen()`** creates both a system window and a graphics context, and binds them together, producing the binding (*context window*) as its return value.

Function **`WinNewContext()`** creates a new graphics context, binds it with the old window, and returns the new context window. The new and old window handles can write to the same system window. Closing either of those window handles removes the window.

Graphics contexts reduce the traffic between the X server and the application programs. The context information is maintained by the X server and only needs to be sent once, and that information is automatically applied to later drawing requests specifying the id of the graphics context. In the case of changes, only the required few fields need to be sent. For example, when setting foreground, only the foreground field needs to be sent; when setting the font, only the font field needs to be sent.

Multiple graphics contexts can be created so application programs can switch among different graphic bindings with ease. A common practice is to create all required context windows at the beginning of resources or processes, set desired attributes for each context window, then use them wherever appropriate. Two examples of the use of graphics contexts and context windows are the button program and Quicksort program discussed earlier, and shown in Appendices B and C.

## Enabling and Disabling On-screen Outputs

SR Win maintains a backup copy of window contents. When a drawing primitive is called, it draws to the backup copy first, then draws to the system window. Both copies are stored in the X server.

To further reduce the communication traffic between the X server and the application program, on-screen drawing can be temporarily disabled by a call to **`WinDisableOutput()`**. This is done on a per top-level window basis. One call on any of the subwindows or context windows will disable on-screen output for the whole window tree. Subsequent drawing will only affect the backup copy. When it is desired the window can be updated by calling **`WinUpdateWindow()`**, which copies the backup copy to the system window on screen. To enable drawing, use **`WinEnableOutput()`**.

## Using Images

For an application that operates on the pixel level, using images could greatly improve performance.

Function **`WinCreateImage()`** creates a memory area for storing an image of specified depth and size. The following code fragment creates an image of size 200 by 100 in pixels using the default depth of the window:

```
var im: winImage := WinCreateImage(mywin, UseDefault, 200, 100)
if im = null ->
  write("Cannot create image")
  stop
fi
```

**`WinPutPixel()`** writes a pixel value to the image. *Pixel values* can be pre-calculated using **`WinSetForeground()`**, which returns an integer value representing the specified color:

```
var p: winPixel := WinSetForeground(mywin, "red")
# other processing
```

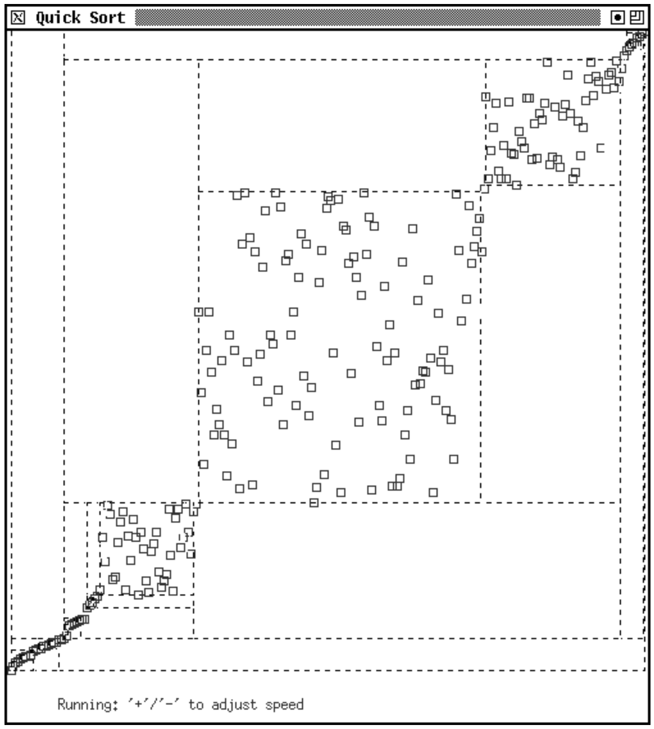


Figure 5. Screen dump of the Quicksort program

data array has been sorted, the administrator process prompts the user for generating a new set of data and starting over again.

The complete program is listed in Appendix C.

# 8. Speeding Up the Program

Transmitting messages over the network takes time and slows down the program. Thus, calls to SRWin that generate network traffic are expensive. This section describes ways to reduce network traffic.

To prevent multiple access to shared X data structures, SRWin employs low-level locking in many function calls. Calling those routines prevents the program from gaining better parallelism. Even if there is another implementation based on a Multi-threaded Xlib [Gild93] available, which can eliminate much locking, the application must not abuse the input and output flows. Keeping these in mind often helps in developing a fast application.

## Reducing Unnecessary Input Events

When an event happens on a window, it is checked against the event mask of the window. If this event is among the selected types, it is passed to SRWin for processing. If it is not, it is passed to the parent window to be checked. By registering only the required event types, an SRWin program can reduce the traffic on the network and, hence, improve performance.

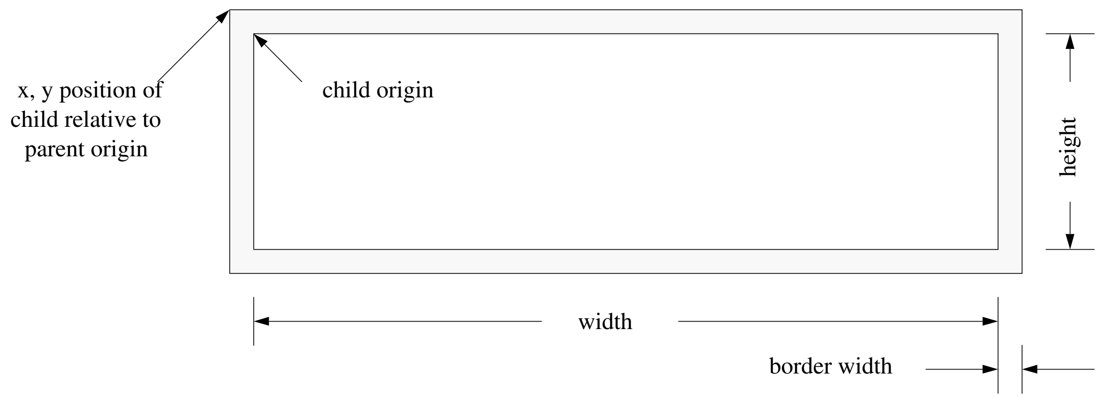


Figure 4. Window Configuration

record type `buttonRec` to characterize a button, an operation type `buttonCallBack` for declaring call-back functions, and an operation `button()` for creating a button. An application program usually "*sends*" to operation `button()`, and the registered call-back function is invoked when the user clicks on the button.

## 7. Visualizing a Quicksort Algorithm

In this section, a Quicksort visualization program, which was inspired by [Brow91], is presented as an example of a larger program.

The Quicksort algorithm uses a few worker processes that run in parallel with a *bag of tasks* to assign work to processes dynamically. Initially, an administrator process generates an array of random data points and it puts the initial task, which is denoted by a pair of indices *(1, n)* for all the points, in the bag. Worker processes then remove tasks one at a time for processing. When a process gets a task *(l, h)*, which marks the region the worker should work on in the data array, it partitions the region according to the traditional Quicksort algorithm. If the partition results in one or more new tasks, the worker process gives itself the first new task for further processing and puts the remaining tasks back in the bag. If the partition results in no new tasks, the worker process tries to get another task from the bag.

The data points are drawn as little boxes on screen. The data array lies horizontally, so that for each data box, its *x* coordinate corresponds to its index or position in the array and its *y* coordinate corresponds to the value of the data. When the array is sorted the boxes should line up from lower-left to upper-right corner.

Each worker process is assigned a unique color. When a worker gets a new task, it changes the color of all points in that region. This makes it easy to distinguish when different processes work on different regions of the data array.

To make the data manipulation more obvious, each worker draws a dashed box around the area it will work on when it gets a new task. This clearly marks each partition (as in Figure 5), and also shows how bad the situation can be when the algorithm chooses a bad pivot point.

The administrator process which initializes the bag of tasks also interacts with the user. For example, the computer would be too fast for human eyes to catch any details of the changes on the data. Thus, there are places where the program delays for a certain amount of time. When the sorting is in progress, the administrator accepts user input to adjust the delay time. In order to detect termination of a sort, the administrator process gathers information about what has been done. When the

```
subw := WinCreateSubwindow(mywin, evchan, UseDefault, 10, 20, 300, 400)
```

The subwindow begins at *(x, y)* relative to its parent's top-left corner. The subwindow can use its parent's event channel, or it may register a new one, or not use any. Subwindows can help a programmer implement common user interface objects such as dialog boxes, buttons, and menus.

Windows do not need to be visible. If **`OffScreen`** is used instead of **`UseDefault`** when creating a window, the window is initially kept off-screen. When it is necessary the window can be "*mapped*" to be displayed. For example, subwindows used as dialog boxes are not always displayed. When the program needs some specific input from the user, a dialog box can be popped up, and after the user has finished input, the dialog box can be hidden away:

```
var dbox: winWindow := WinCreateSubwindow(mywin, evchan,
                                          OffScreen, 10, 10, 100, 100)
# some processing ...
WinMapWindow(dbox)
# asks user input, read, some processing ...
WinUnmapWindow(dbox)
```

However, some other things need to be done to make this code work. For each window, there is an *event mask* attribute that tells what kind of events this window is expecting. When an event is generated, it is tested against the event mask of the window. If the event is of selected type, it is sent to the registered event channel. Otherwise the event is passed to the parent of the window and checked again. When a window is created initially off-screen, the event mask is empty. If afterwards input on this window is desired, SR Win routine **`WinSetEventMask()`** can be used to select events of interest:

```
WinSetEventMask(dbox, Ev_KeyDown | Ev_ButtonDown)
```

The event mask is set to a bitwise *or*'ed list of event types. In the above example, future key and mouse button press events will be delivered to the event channel for that window.

Take a button as another example. A software button may look like a rectangular box. It may change its appearance when the user presses a mouse button within it.

```
open subwindow
set initial border and event mask, write label
map window
do true ->
    get event
    if button pressed ->
        change to reverse video
    [] button released ->
        change to normal display with highlighted border
        invoke the user specified call-back function
    [] enter window ->
        highlight border
    [] exit window ->
        back to normal display
    fi
od
```

When mapped on screen, a window may have a border surrounding it. The window border may be used for drawing attention to a special area, for example a button. However the border is different from the decoration that a window manager would put for each window. When creating a window, the width of the border is not included in the size of the window. When placing a subwindow, its border needs to be taken into consideration (see Figure 4).

SR Win function **`WinSetBorder()`** sets the border width and border color of a window, e.g.:

```
WinSetBorder(mywin, 3, "red")
```

A sample program that implements buttons as described above is listed in Appendix B. It includes a global `Button` and a test resource that shows how to create a button. The global `Button` exports a

```
resource pointer()
  import SRWin                          # first thing
  const W := 500
  const H := 300
  # open window and load font
  op evchan: winEventChannel
  var mywin: winWindow := WinOpen("", "Pointer", evchan, UseDefault, W, H)
  if mywin = null ->
    write("Sorry, cannot open a window")
    stop(1)
  fi
  var ft: winFont := WinLoadFont(mywin, "lucidasanstypewriter-14")
  if ft = null ->
    ft := WinDefaultFont(mywin)
  [] else ->
    WinSetFont(mywin, ft)
  fi
  # interact with the user
  do true ->
    var ev: winEvent
    receive evchan(ev)
    if ev.event_type = Ev_PointerMove or ev.event_type = Ev_EnterWindow ->
      var str: string[100]
      sprintf(str, " Pointer is at: (%3d, %3d) ", ev.x, ev.y)
      var strWidth: int := WinTextWidth(ft, str)
      WinDrawImageString(mywin, winPoint((W - strWidth) / 2, H / 2), str)
    [] ev.event_type = Ev_ExitWindow ->
      WinEraseArea(mywin, winRectangle(0, 0, W, H))
    [] ev.event_type = Ev_DeleteWindow or
        (ev.event_type = Ev_KeyUp and char(ev.data) = 'q') ->
      exit
    fi
    WinFlush(mywin)
  od
  WinClose(mywin)
end pointer
```

Figure 3. The Pointer Tracking Example

Another common programming model is the de-coupled programming model, where some processes do the computation while a coordinator process communicates with the computing processes and interacts with the user. It is best illustrated in Section 7 (“Visualizing a Quicksort Algorithm”).

# 6. Advanced Window Management

Windows created by **`WinOpen()`** are often called *top-level* or *application level* windows. They are managed by a special program, the *window manager*. SR Win supports yet another kind of windows, *subwindows*. Hierarchically, a subwindow is a child of another window, which is referred to as its parent. When displayed, a subwindow is placed on top of its parent and *clipped* inside its parent. Subwindows share a single network connection to the display with their parents. A top-level window may have several subwindows as its children. A subwindow may also have its own subwindows.

Function **`WinCreateSubwindow()`** creates a subwindow as a child of an existing window:

```
var subw: winWindow
# starts at (x, y) = (10, 20) in the parent window, width 300, height 400
```

The goal of the program is to keep track of the pointer position in a window it opens and let the user know where the pointer points to. This can be outlined as:

```
open window
do true ->
        read event
        if pointer enters the window or pointer moves ->
                write the coordinates of the pointer in the center of the window
        [] pointer leaves the window ->
                clear the window
        [] should quit ->
                exit
        fi
od
close the window
```

Opening a window, reading events, and closing the window are similar to those in the previous sections.

Event types **Ev_PointerMove**, **Ev_EnterWindow**, and **Ev_ExitWindow** indicate state changes of the pointer. The *x* and *y* fields in the event record give the coordinates of the pointer at the time the event occurs:

```
var ev: winEvent
# read event ... it is Ev_PointerMove or Ev_EnterWindow
var str: string[100]
sprintf(str, "Pointer is at: (%d, %d)", ev.x, ev.y)
# write string "str" in the center of the window ...
```

One way to center the string in the window is to compute the width in pixels of the string, then calculate how far the string would be from the left edge of the window — that is, where the string should start. The width of the string depends on the window's current font. The same string displayed in different fonts may have different widths.

Fonts are usually represented by bitmap data stored in files. A program may have several fonts *loaded*, so that when it needs a particular font for displaying characters, it can quickly switch to that font. There are some standard X Window utilities, such as **xlsfonts**, **xfontsel**, etc., which list or view various fonts. The SRWin function **WinLoadFont()** loads a font:

```
var ft: winFont := WinLoadFont(mywin, "lucidasanstypewriter-14")
```

If the specified font cannot be loaded, **WinLoadFont()** returns `null`. Since a window starts up with a default font, a program can always use **WinDefaultFont()** to find out the default font:

```
if ft = null ->
  ft := WinDefaultFont(mywin)          # will always get the font
fi
```

To set a font on a window for subsequent text output, a program calls **WinSetFont()**:

```
WinSetFont(mywin, ft)
```

Once a font is loaded, **WinTextWidth()** will give the width of a string in a specific font, which will help, for example, to center a string:

```
var strWidth: int := WinTextWidth(ft, str)
# suppose "W" is the width of the window, "H" is the height of the window
WinDrawString(mywin, winPoint((W - strWidth) / 2, H / 2), str)
```

The complete program for the "pointer tracking" example is listed in Figure 3. On some configurations of the X Window System, an **Ev_DeleteWindow** event is generated when the user chooses "Close" or "Quit" entry from a window manager menu. The example program presented here uses this information with keyboard input "q" to decide when it is about to exit.

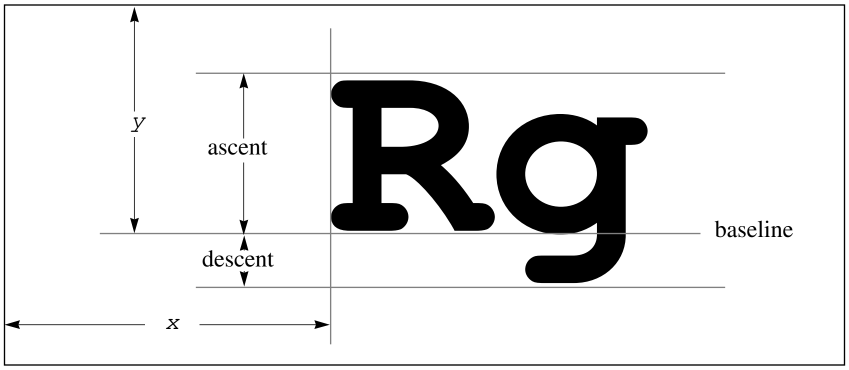


Figure 2. Position of a string drawn at *(x, y)* in a window

```
op evchan: winEventChannel
var mywin: winWindow := WinOpen("", "Test", evchan, UseDefault, 100, 50)
```

Upon successful creation of a window, a process is set up to poll and dispatch window events to the message channel using asynchronous message passing. The event polling process is called an *event handler*. The message channel is called an *event channel*. An event is represented by a record **`winEvent`**. A program can determine whether there is an event pending by checking the number of messages in the event channel:

```
if ?evchan > 0 ->
  write("Something happened on that window")
fi
```

Or the program can wait on the event channel until an event happens:

```
var ev: winEvent
receive evchan(ev)
```

Upon receiving an event, a program can check the *`event_type`* field of the event record to see what kind of event has just happened. For example, there are event types **`Ev_KeyDown`** and **`Ev_KeyUp`**, which stand for key press and key release events respectively. In these cases the *`data`* field of the event record is the ASCII code of the key:

```
var ev: winEvent
# ... get event ...
if ev.event_type = Ev_KeyDown ->
  write("Key pressed: ", char(ev.data))
[] ev.event_type = Ev_KeyUp ->
  write("Key released: ", char(ev.data))
fi
```

More uses of event channels are shown in the next section with an example program.

# 5. Programming Models

One common way to build an interactive graphics program is the *event-driven* programming model. An event-reading loop is the primary control mechanism driving the program — the program reads some input, does some processing and drawing, then comes back to read input. It can be illustrated by the following example program.

and the $Y$ axis pointing down. The **`WinDrawString()`** statement draws the string close to the bottom-left corner of the window.

After the program finishes processing, it calls **`WinClose()`** to clean up. This concludes this small working SRWin program.

## 3. Output

The "Hello World" program illustrates the SRWin conventions for naming and syntax of its data structures and routines. All SRWin functions begin with "**`Win`**" followed by compound words constructed by capitalizing the first letter of each word (e.g. **`WinOpen()`**, **`WinFlush()`**); data types are constructed similarly except that they begin with "**`win`**" (e.g. **`winWindow`**, **`winPoint`**).

Usually, **`WinOpen()`** is the first SRWin routine that a program invokes. It returns a window handle, which then can be used as the first argument to many drawing routines. A program calls those drawing routines to display information on the window. For example, a statement that draws a line would look like:

```
WinDrawLine(mywin, winPoint(0, 100), winPoint(100, 0))
```

The type **`winPoint`** converts the $x$ and $y$ coordinates to a point.

SRWin provides a set of routines to draw lines, ellipses, texts, and individual pixels inside a window. Some routines can do filling. Multiple characteristics of drawing and filling, such as *line width* and *fill style*, can be controlled. Taking lines as an example, *line style* specifies whether a line should be drawn solid or dotted, etc. *Cap style* tells whether the endings of a line should be rounded or not. *Join style* sets whether the joint of two segments in a polyline should be rounded or not. The following two statements draw a 20 by 10 rectangle with dashed lines:

```
WinSetLineAttr(mywin, 0, LineDoubleDash, CapButt, JoinMiter)
WinDrawRectangle(mywin, winRectangle(1, 1, 20, 10))
```

Here the line width is set to zero in order to use a fast algorithm for line drawing, while the line actually being drawn has width 1 in pixels. If the line width is greater than zero, a more general, hence slower, algorithm will be used.

Drawing is not limited to only one window for a program. Multiple windows can be created using **`WinOpen()`**. Windows can be opened on different machines by explicitly specifying the machine as the host part of the display parameter:

```
var mywin1: winWindow := WinOpen("can:0", "1", null, UseDefault, 100, 50)
var mywin2: winWindow := WinOpen("gly:0", "2", null, UseDefault, 100, 50)
```

A string of characters is positioned as depicted in Figure 2. **`WinDrawString()`** draws the characters. **`WinDrawImageString()`** clears the extent of the string first, then draws it. In this way the string will not be cluttered by previous drawing at the same location.

Sometimes when a region of a window becomes obscured by other windows, the contents inside that region are lost. When the obscured region becomes visible again, the X application must redraw that part of the window. SRWin, however, handles this automatically, refreshing the display from a backup copy of window contents.

## 4. Input

A window is not only capable of displaying output, it is also capable of reading input. Inputs from a window are modeled as *events*. The **`SRWin`** global exports an optype **`winEventChannel`** which is used by an application program to declare a message channel for accepting events. The SR program passes this when opening the window:

# 1. Introduction

This document is intended for programmers writing graphics programs using SRWin, a library of graphics facilities for the SR programming language [AnOl93]. It describes the library and presents some examples of its use. Appendix A gives some details about the implementation of SRWin and some thoughts for further development. Appendix B gives a program that demonstrates window buttons. Appendix C lists a program that visualizes a parallel Quicksort algorithm.

SRWin adds an interface to graphics primitives provided by the X Window System [ScGe86]. It is based on Xlib [NyeA92, ScGe92], the low-level C interface for X Window programming. Given the availability of the underlying X Window System, SRWin can run on many platforms. Given the SR concurrent programming environment, and a few abstractions for user-computer interaction, SRWin becomes simple, easy to program, and effective in the sense that even naive programmers may develop fairly efficient graphics applications.

# 2. A “Hello World” Program

Writing graphics programs in SR is not too complex. A simple SRWin program might look like this:

```
resource hello()
  import SRWin
  var mywin: winWindow := WinOpen("", "Hello", null, UseDefault, 100, 30)
  if mywin = null ->
    write("Sorry, no window.")
    stop(1)
  fi
  WinSetForeground(mywin, "yellow")
  WinDrawString(mywin, winPoint(10, 20), "Hello World")
  WinFlush(mywin)
  nap(10000)
  WinClose(mywin)
end hello
```

Figure 1. The “Hello World” Program

This program imports a global **`SRWin`** and is linked with a file of associated C functions and the X library. During execution, it simply opens a window on the default display, writes “Hello World” in yellow, waits for ten seconds, then exits.

There is an analogy between a window in SRWin and a file in SR. SRWin uses **`WinOpen()`** to open a window, which in turn returns a window handle. SR uses **`open()`** to open a file, which returns a file handle. Like its counterpart for file access, **`WinOpen()`** returns `null` if the window cannot be opened. SRWin also uses buffered I/O. While **`WinDrawString()`** writes to the window just as **`write()`** writes to a file, the result may not appear immediately in the window. **`WinFlush()`** acts similarly to **`flush()`**: it flushes the output buffers of the window. Finally, **`WinClose()`** closes a window just as **`close()`** does for a file.

Specifically, **`WinOpen()`** first tries to establish a connection to the display named by its first argument. In the above example the display parameter is an empty string, so the default specification stored in UNIX environment variable `DISPLAY` is used. Then the function opens a window and uses the second argument as its title. The width and height of the window must be provided as the last two parameters. The other parameters will be discussed later.

Once the window is created, the program can call SRWin functions to draw on the window. There is an integer coordinate system on each window with the origin at the top-left corner of the window

# SR Win
## A Graphics Library for SR

Qiang Alex Zhao

TR-93-14


**Abstract**

This document describes the calling interface and usage conventions of SR Win, a graphics library for the SR concurrent programming language. SR Win provides a simple environment for building interactive graphics system. It currently runs on UNIX under Version 11 of the X Window System.


May 1, 1993

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

# SR Win

## A Graphics Library for SR

Qiang Alex Zhao

TR 93-14

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF ARIZONA

TUCSON ARIZONA