WIN: A Persistent Window Management System

Q I Cutts*, A Dearle*, G N C Kirby* and C D Marlin†

*University of St Andrews, North Haugh, St Andrews, Scotland KY16 9SS.
†University of Adelaide, GPO Box 498, Adelaide, South Australia 5001, Australia.

Abstract

WIN is a persistent window management system written in Napier88, a strongly typed persistent language developed at the University of St Andrews. The system treats windows as independent objects which may be displayed by a window manager or retained in a persistent store as required. Because windows are not tightly bound to specific window managers, they may be reused: windows may outlive a particular invocation of the WIN system.

Window managers themselves run within windows and so they may be nested to arbitrary depth. This enables a hierarchical arrangement of windows, where a window may in turn be organised by a window manager which itself displays sub-windows. The benefits of this approach are that applications which run within windows may be easily structured, and that user interface tools such as light-buttons, menus and sliders are treated uniformly as windows.

This document should be referenced as:
"WIN: A Persistent Window Management System
Universities of St Andrews and Glasgow, PPRR-73-89"

1. Introduction

The window manager paradigm has become widespread, and it has been used to provide user interfaces for a wide range of applications and systems. These have included operating systems, applications such as editors, programming environments, and CAD systems. Some examples of window management systems are the Macintosh system [1], SunView™ [2], Sapphire [3], and the Layers system [4].

This paper describes a window management system written in a high level, strongly typed language, Napier88 [5], designed as a tool available to programmers writing systems within the Napier88 environment. Features of the system include the support of persistent windows, the hierarchical nesting of window managers, and the use of an event-driven mechanism for distributing input events to windows.

1.1 Persistence

Napier88 supports orthogonal persistence [6,7]; that is, the length of time for which an object persists is independent of its other attributes. This length of time can vary from relatively short, as with variables local to a program block, to relatively long, as with data held within a database system.

In systems without orthogonal persistence there are traditionally two levels of data representation: a structured form of the data, created and manipulated within programs, and another form used by the operating system to store the data for lengths of time greater than a single program execution. This second form is normally more restricted than the first. For example, UNIX provides only one-dimensional byte strings, files, for long-term storage. Database systems may allow greater structure to be imposed on long-lived data, but this structure is typically still less rich than that used within programs.

There are significant programming and systems overheads in translating between these two forms; these overheads disappear when a persistent system is used. In a persistent system, the data is stored in its structured form, so that no reconstruction process is necessary when a program accesses data held in the store.

1.2 Persistent Windows

The idea of orthogonal persistence is rarely extended to the window manager paradigm. In most window management systems, a new instance of a window manager is created each time the system is invoked. The window manager provides the user with the capability to create windows which may be manipulated using that window manager. These windows are tightly bound to that specific

instance of a window manager, meaning that they are associated with the window manager, and if that is subsequently discarded then the windows will automatically be lost along with it.

In the WIN (Windows In Napier88) system, windows are independent objects (abstract data types), which are created by a separate window creation procedure rather than by a window manager. Window managers provide a mechanism whereby these objects may be viewed. When a window is displayed by a window manager it may become visible to the user; however, all the window's operations may be accessed irrespective of whether or not it is displayed.

An advantage of divorcing windows from window managers in this way is that windows may be moved between window managers, or kept in the persistent store and later recalled and displayed. In the current implementation, a window cannot be displayed by more than one window manager simultaneously – a consequence of the method of implementation, discussed in Section 8. However, a window may be undisplayed by one window manager and then displayed by another.

The ability to make windows persistent, outliving particular window manager instantiations, allows the programmer using the system to reuse windows. Some examples of windows which could be reused are pop-up menus, mail windows and editor windows.

1.3 Napier88

The system is implemented entirely in Napier88. The language is well suited to the construction of large software systems, with its first-class procedures, its simple but powerful type system, and its persistence facilities which enable the programmer to build software from small modules. It also supports raster graphics which are used extensively throughout the WIN system.

WIN is the first large system to be written in Napier88, other than the Napier88 compiler itself.

2. Overview

This section aims to give an overview of the WIN system, which was developed from an earlier window management system written in PS-algol [8], a predecessor of Napier88. In that system, called PStools [9], windows were not treated as independent objects, but were bound to particular window managers. This meant that it was impossible to reuse windows, and a number of applications could have been written more easily if this had been possible. The frustrations caused by this provided the initial motivation for the development of the WIN system.

2.1 Windows

An image in Napier88 is a rectangular grid of pixels, and the language provides operations for performing raster operations between images, and for making aliases onto parts of images. The screen is treated as an image, so rastering onto the screen is carried out in the same way as other raster operations.

A window in the WIN system is regarded as an abstraction over an image, in that it has a rectangular area onto which raster operations may be performed. Each window encapsulates an image, and it also has an associated **application**. An application is a procedure which takes an input event as its parameter, and performs some action; every application has this same fixed interface. In the current WIN system, the input event may be either a string of characters from the keyboard, or a structure giving the state of the mouse (i.e. its position and which of its buttons are pressed). In Napier88, this is described as follows:

```
type Mouse is structure ( x,y : int ; buttons : *bool )
type Event is variant ( chars : string ; mouse : Mouse )
type Application is proc ( Event )
```

The first type declaration specifies the type *Mouse* as being a structure, or record, containing two integer values which give the position of the mouse, and a vector of boolean values. Each element of this vector, has the value true if the corresponding mouse button is depressed, and the value false otherwise.

The second declaration specifies *Event* as a variant type, meaning that a value of this type may be of either type *string* or *Mouse*, with the label names distinguishing between them. For example,

```
let a = Event( chars : "hello" )
let b = a'chars
```

would result in the value "hello" being bound to the identifier *b*.

Finally, the declaration of *Application* defines values of this type to be procedures which take one parameter, of type *Event*, and do not return any result.

The interface to a window in WIN is a package of procedures, each of which performs some window operation. These operations include:

- raster operations between the window and another window, and between the window and an image,
- line drawing on the window,
- changing the size of the window,

- setting the application for the window, and
- setting the title of the window.

The interface is described more fully in Section 4.

2.2 Window Managers

A WIN window manager is an object which organises and displays a set of (possibly overlapping) windows. When a window manager is created, it is given a window in which to operate. A consequence of this is that any window may have a window manager running within in it, so window managers may be nested to any depth. This is useful for organising the contents of a window: a window manager may be used within it to display items such as light-buttons, scroll-bars and sub-windows.

The system provides a procedure to create a window operating directly on the screen; a window manager at the top level will work within such a window.

Like a window, a window manager is presented as a package of procedures. These include procedures for:

- displaying a window at a given position and depth (the degree to which it obscures or is obscured by other windows) relative to other windows,
- undisplaying a window,
- moving a window, and
- closing and opening (iconising and restoring) a window.

To resolve the question of which window should receive input events, the WIN system has the notion of the **current** window. At any time either one window or none is current. If none of the windows displayed by a window manager is current then events detected by the window manager are discarded. The current implementation has 'follow-mouse' behaviour: the user is given the illusion that all windows are ready to receive mouse input. This is achieved by having the system make a window current whenever the cursor is moved over it. However, this could easily be changed to give 'click-to-type' where the previous window remains current until the user clicks a mouse button while the cursor is over another window. The user of the system can also control which window is current: there is a window manager procedure to make a given window current.

Applications do not actively poll for input events, but receive events from a central agency, with which they have previously registered a description of the kinds of events in which they are interested. This has the advantage that an application in one window will not receive and discard events intended for an application in a different window. This event distribution system is described in Section 3.

2.3 Independent Windows

In the WIN system, a window is not necessarily displayed by any window manager. It is created independently, and will not be displayed unless the display procedure of a window manager is called with the window as its parameter. When the window is not displayed, it does not receive any input events through the window manager. However, there is nothing to prevent the procedure which is the window's application being called from outside the window manager.

If the window becomes displayed, then changes to it will be visible if the window is not obscured by others, and the window may receive input events. When the user directs some input to the window, the event is first sent to the window manager displaying it, and from there distributed to the appropriate window.

3. Event Distribution

WIN uses a central event distribution system [10]. Events are gathered by a central **event monitor**, and then distributed via a **notifier** hierarchy. This mechanism is a general one and is not solely specific to window management systems.

3.1 Event Monitors

An event monitor is a procedure containing a loop which continually polls for user input. From the input it constructs events, which may contain either strings of characters from the keyboard or information about the state of the mouse. When an event has been formed the event monitor calls its notifier (described below) to distribute it to the appropriate application.

3.2 Notifiers

A notifier is an object which receives events and then distributes them to one of a number of applications which have been previously registered with it. It has two components: a procedure to register applications, and a procedure to distribute events among the registered applications.

The interface to a notifier is as follows:

```
type Notification is structure (          examineEvent :   proc ( Event → bool );
                                  processEvent :   Application )

type Level is structure ( fromFront : bool ; position : int )

type Notifier is structure (          distributeEvent :   Application;
                                  addNotification :   proc ( Notification,Level → proc() ) )
```

To register an application with a notifier, the *addNotification* procedure of the notifier is called. This takes a **notification** and a **level** as parameters. The notification contains two procedures. One is the application itself; the other procedure takes an event as parameter and returns a boolean value. This procedure, called the *examineEvent* procedure, is used to determine whether a given event is suitable for the application. The notification is placed in a list within the notifier, at a position specified by the level. A procedure is returned which, when called, will remove the notification from the list.

When the notifier is called upon to distribute a particular event (i.e. when its *distributeEvent* procedure is called), it searches down its list, calling the *examineEvent* procedure for each notification with the new event as its parameter. It continues until one of the procedures returns the value true, or until there are no more notifications. If one of the applications accepts the event, it is then called with the event as its parameter. More than one application may wish to accept the event; in this case, it is the one in the highest position in the list which receives the event.

The level at which a notification is placed is specified in terms of a position from either the top or the bottom of the list. Level(true,1) would put the notification at the top of the list, whereas Level(false,2) would put it second from the bottom.

3.3 Nesting Notifiers

Notifiers may be nested, as the *processEvent* field of *Notification* and the *distributeEvent* field of *Notifier* are both of type *Application*. Nesting is achieved by supplying the *distributeEvent* procedure of one notifier as the application of a notification registered with a higher level notifier. This enables the construction of notifier hierarchies, where a top-level notifier decides which category an event lies in, and then passes the event to another notifier for distribution within that category. This structure can be generalised to any number of levels.

This is useful as window managers may also be nested. When a window manager is created, it is given a new notifier with an empty notification list. Each time the window manager displays another window, it registers a notification for the window with its notifier. The *examineEvent* procedure in the notification tests mouse events to determine whether they occurred over a visible

part of the window. Keyboard events are always received by the current window. If the *examineEvent* procedure returns true, then the *processEvent* procedure calls the application for the window.

When a window manager is created the window in which it is to run must be specified. If this window is already displayed by another window manager, the *distributeEvent* procedure of the notifier of the new window manager becomes the application of that window. The result is a notifier hierarchy which maps directly to the hierarchy of window managers.

This is illustrated in Figures 1(a) and 1(b). Figure 1(a) shows a hierarchical arrangement of window managers: windows ❷ and ❸ are displayed by a window manager running in window ❶, and each contain a window manager themselves. There is a further window manager running in one of the windows displayed by window ❷'s window manager.

Figure 1(b) shows the hierarchy of notifiers corresponding to these window managers. There is a top-level notifier used by the window manager running in window ❶. Two of its notifications contain the *distributeEvent* procedures of notifiers ❷ and ❸. Notifier ❷ in turn has notifier ❹ below it, reflecting the fact that the window manager within window ❷ displays the window within which window manager ❹ operates.

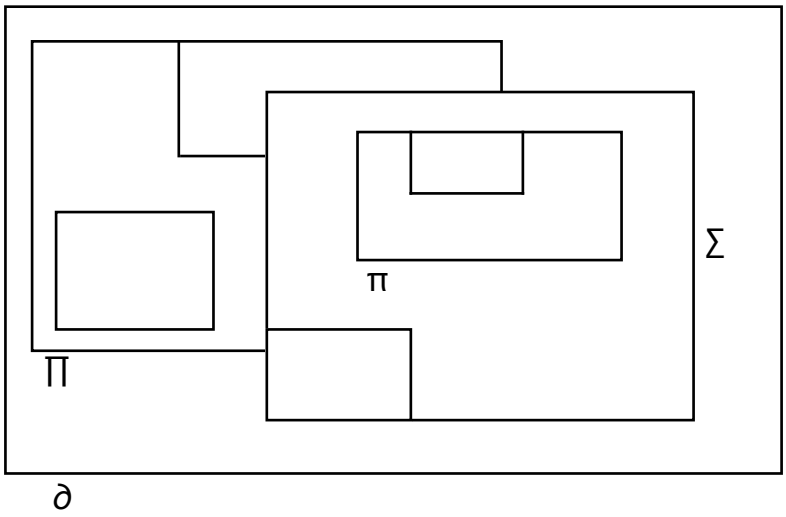

Figure 1(a). A Hierarchy of Window Managers

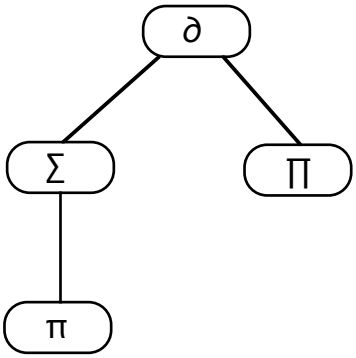

Figure 1(b). A Corresponding Hierarchy of Notifiers

4. Windows

It has already been emphasised that windows are independent objects which may be manipulated and stored, whether or not they are displayed by a window manager. This section describes how windows are created, and what operations may be performed on them. Windows are described in greater detail in [11].

4.1 Making Windows

Windows are generated using the procedure

windowCreator : **proc**(Dimensions,Dimensions,Dimensions → Window)

where the type *Dimensions* is used to describe the size of a window, and is defined by

type Dimensions **is structure**(sizeX,sizeY : int)

This means that *windowCreator* is a procedure which takes as its parameters three instances of the type *Dimensions*, and returns an instance of type *Window*. The three parameters refer to the initial, minimum and maximum sizes of the window. This enables the creator of the window to prevent it being subsequently resized beyond the specified limits. This is useful when the programmer wishes to make sure that the window will not become too small for its contents, which may include light-buttons, sub-windows, etc.

4.2 Window Procedures

Window is a structure type which includes the procedures described below:

- *windowRasterTo* and *windowRasterFrom* enable raster operations to be carried out to the window from another and vice-versa;
- *imageRasterTo* and *imageRasterFrom* perform similar functions, but with simple images rather than windows – these are provided so that raster operations over small areas may be carried out without the expense of creating a new window;
- *drawLine* is used for drawing lines onto windows;
- *getSize* returns the size of the window;
- *resize* alters the size of the window, subject to any minimum and maximum size constraints;
- *setApplication* and *getApplication* respectively set and return the application associated with the window;
- *setTitle* and *getTitle* respectively set and return the title of the window;
- *setResizeProc* sets a procedure which is executed whenever the window is resized;
- *takeInput* sets the window's status regarding its reception of input events – it may choose to receive all events occurring, none of them, or just those normally sent to it – see Section 5.3;
- *getWindowManager* returns the window manager which is currently displaying the window, if any.

5. Window Managers

Window managers control the display and appearance of windows on the screen. This section describes how they are created, what the interface to a window manager provides, how window managers distribute input to windows, and how borders are used to allow interactive manipulation of windows.

5.1 Making Window Managers

Window managers are created using the procedure

windowManagerCreator : **proc**(Window,Notifier → WindowManager)

which initially takes a window within which the window manager will operate, and a new notifier, and then returns a new window manager.

5.2 Window Manager Procedures

WindowManager is another structure type containing a number of procedures. The principal procedures are described below:

- *display* takes a window and displays it at a given position and at a given level relative to the windows already present;
- *undisplay* removes a window from the window manager's display area, leaving the window itself unaffected;
- *move* and *putAtLevel* allow a window to be moved both in position and level;
- *close* and *open* respectively replace a window by its icon, and restore the window again;
- *makeCurrent* makes the given window the current window, so that it may receive input events;
- *setIcon* and *setIconPos* set the appearance of a window's icon, and the position at which the icon will appear;
- *getPos* returns the position of the origin of a given window;
- *setBackgroundApp* sets an application to run in the background of the window manager – input events which are not picked up by any window are sent to this application;
- *redraw* regenerates the window manager's display area, which is useful if the window containing the window manager is resized.

5.3 Receiving Input

As described earlier, the current window is the only one which receives input events through the window manager. It is sometimes useful for the current window to be able to 'grab all the input', so that it receives all input events occurring, whether or not they would normally go to that window. An example of this is the displaying of a confirmation box, where the application programmer wishes to force the user to reply to a query before continuing with any other activity. The application can do this by grabbing all the input until the user has replied, and then restoring the window to its normal state.

The window procedure *takeInput* provides this facility; it takes as its parameter a value of the following type:

type InputOption **is variant** (all,none,normal : null)

in which the labels are used to give the effect of an enumerated type. When the procedure is called with InputOption(all : nil), where *nil* is the single value of type *null*, the window receives all the input events which are detected by its window manager, until a subsequent call of the procedure with the value InputOption(normal : nil). A third value, InputOption(none : nil), allows the window to ignore all events.

If the procedure is called when the window is not current, there is no effect until it does become current.

5.4 Borders

Each window is displayed with a border around it, serving to show the outline of the window and whether or not it is current; the border also allows the user to manipulate the window interactively. A typical border is shown below in Figure 3. The fact that the area in the bar at the top is grey indicates that the window is current; when the window is not current this area is white. The title of the window is also shown in this area. The left-hand box within the border allows the user to iconise the window, and the right-hand box allows it to be resized. The window can also be moved by dragging on the bar surrounding the boxes.

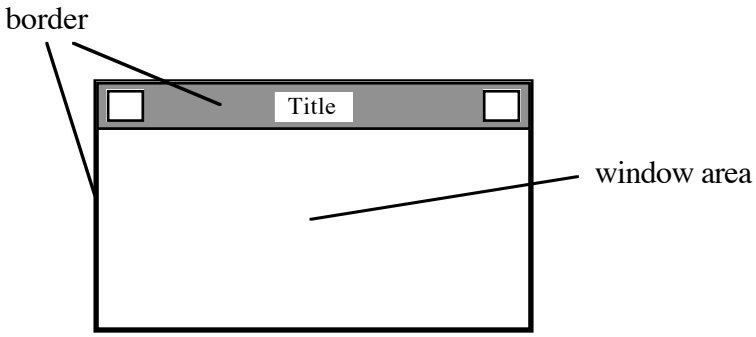

Figure 3. A Window Border

This is just one possible style of border. WIN has a number of border styles which are predefined, but the user is also free to define other styles, giving flexibility as to both the appearance and the functionality of the borders displayed.

A border style is defined by splitting it up into a number of areas:

```
type Pos is structure( x,y : int )

type Area is structure (   currentImage,nonCurrentImage : image ; pos : Pos ;
                           distributeEvent : Application )

type BorderStyle is proc ( Window → List[ Area ] )
```

List is a parameterised list type: a border style is represented as a procedure which takes as its parameter a window and returns a list of values of type *Area*. Each of these is a structure containing two images, one for when the window is current, and one for when it is not. The structure also contains the position of the origin of the area relative to the origin of the window, and a procedure which takes an input event as its parameter. This procedure determines whether the

event occurred over a part of the border area with a procedure associated with it – such as the white boxes in the style shown above – and if so calls that procedure.

When the window manager receives an event over one of the border areas of a window, it calls the application from the corresponding *Area* structure. This applies to all windows displayed, not just the current window; consequently, windows can be moved, resized, etc., without making them current. The border style shown could be represented by the four areas in Figure 4.

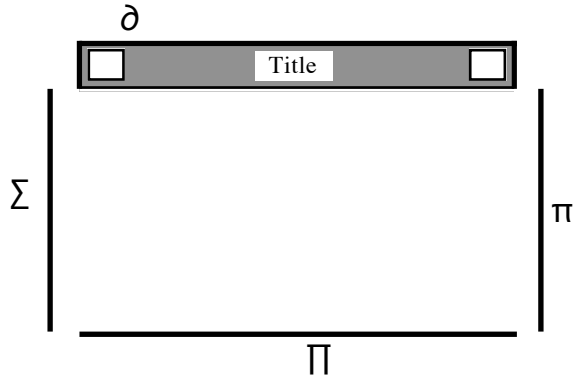

Figure 4. Division of a Border into Areas

A new border style is made by constructing a procedure of the type described, which takes a window as its parameter and returns a list of border areas. The procedure will use the size of the window to calculate the sizes and positions of the areas.

5.5 Interactive Window Manipulation

Windows can be moved, resized and otherwise manipulated by calling the appropriate procedure of the window or window manager, but frequently the user requires some way of doing this interactively. A number of procedures are available for this purpose. The move procedure, for example, displays an outline of the window chosen, and moves it around following the position of the mouse until the mouse button is released. When this occurs, it calls the move procedure provided by the window manager to move the window to its new position.

Each of the interactive procedures is of the following type:

type InteractiveProc **is proc**(Window,Event,**proc**(Event → bool))

The procedures take as their parameters the window to be manipulated, the event which caused the procedure to be invoked, and a procedure to determine whether a given event represents a mouse button press. This last parameter gives flexibility over which of the mouse buttons is used in manipulating the window. The store contains procedures to perform moving, resizing, deleting, closing and exposing windows. The user is free to add to this collection if necessary.

These procedures can be incorporated into a border style by using them within the *distributeEvent* procedures for the border areas. In the style shown in Figure 3 for example, the procedure for the large area determines whether the event received occurred within one of the boxes or in the rest of the area, and then calls either the move procedure, the close procedure or the resize procedure as appropriate. This mechanism allows new border styles to have whatever functionality is desired.

5.6 Background Windows

It is useful for the user to be able to draw onto the background of a window manager. In one example, an application showed links between windows by drawing lines between them on the background. If this were carried out, however, by simply drawing on the window containing the window manager, the contents of the windows displayed might be corrupted.

This facility is provided safely by the ability of a window manager to display a window in the background. Only one window may be so displayed at a time, and a background window is always behind all other windows, no matter what levels they are placed at. If the user wishes to be able to draw all over the background of the window manager, this can be done by creating a window the same size as the window containing the window manager, displaying it in the background, and then rastering to that window.

The *display* procedure of a window manager takes a parameter to specify whether the window should be displayed as normal, or in the background.

5.7 The Background Application

At times, the user may wish to have an application running in the background of the window manager, but not wish to draw onto the background. If a background window is used, there is an unnecessary memory overhead involved in storing the contents of the window. A window manager procedure, *setBackgroundApp*, allows the user to run an application in the background without creating an unnecessary window.

The procedure takes an application as its parameter. Subsequently, any events received by the window manager which are not sent to window applications or border areas are sent to this background application. An example of a situation where this is useful is in the provision of a background menu, where a menu of options is displayed if the user presses a mouse button over the background part of the window manager's display. The background application would simply display the menu at the position specified by the event received.

6. Interface Tools

WIN provides a number of basic user interface tools. These include procedures for creating light-buttons, sliders, menus and scrolling menus. Each procedure takes a number of parameters describing the layout of the interface item, and returns an instance of the following type:

```
type InterfaceItem is structure( itemWindow : Window ; setItem : proc( any ) )
```

The structure contains a window whose application implements the interface tool, and a procedure which may be used to control the setting of the tool. This might be used, for example, to give a slider an initial value on its creation.

Type *any* is the type of the union of all values in Napier88. The *setItem* procedure takes a parameter of this type because the type of the value required depends on the kind of tool: a slider will require a real value, a menu an integer value corresponding to the entry to be selected, and if a new variety of tool is created, it might be set by values of any arbitrary type.

Values of any type can be injected into type *any* using the syntax shown in the following examples:

```
any( 3 )        any( "hello" )
```

An example of projection from any is as follows:

```
let setSlider = proc( a : any ) ! this is in the structure returned on creating a slider
begin
    project a as newValue onto
    real : set the slider to newValue
    default : raise an error ! value injected into any wasn't a real
end
```

6.1 Example: Light-buttons

As an example of the way that interface tools are used, light-buttons will be described. The use of the other tools is discussed in detail in [11].

The procedure to create a light-button is of the following type:

```
proc( Appearance,proc( int ) → InterfaceItem )
where
type Appearance is variant( anImage : image ; aString : string )
```

It takes either an image or a string to display on the button, and a procedure which will be executed when the button is pressed. This procedure is itself passed the number of the mouse button which

was used to press the light-button; thus, the light-button can have a different effect depending on the mouse button used.

The *setItem* procedure returned in the structure makes the light-button flash once, whatever the value passed to it.

Once a light-button has been created, it can be displayed by a window manager in the normal way. It will subsequently function without any further intervention from the applications programmer: whenever the user presses and releases a mouse button with the cursor over the light-button, the procedure associated with it will be executed.

The light-button can be removed and displayed again elsewhere, exactly as any other window can. Because it is implemented as a window, it may outlive the execution of the program in which it was created.

The example below shows how a light-button might be created and used. In this case a word is written out each time the light-button is pressed, the word depending on which mouse button is used.

```
let writeWord = proc( n : int )
      write( case n of
                1 : "alpha"
                2 : "bravo"
                default : "charlie" )

let newButton = createButton( Appearance( aString : "word" ),writeWord )
! The light-button hasn't been displayed anywhere yet.

! Assume windowManager has already been declared.
windowManager( display )( newButton( itemWindow ),Pos( 100,100 ),Level( 1,true ) )
! Displays the light-button at the top level at position (100,100).

! The light-button could be made to flash as though it had been pressed by calling
! newButton( setItem )( any( 0 ) ).
```

6.2 Extending the Tool Set

As is the case with window border styles, the set of user interface tools is designed to be extensible by the applications programmer. A programmer wishing to add a new kind of tool should write a procedure which takes parameters describing a particular instance of the tool and returns an instance of the *InterfaceItem* type described earlier. The procedure could then be placed in the persistent environment, along with tool generators already written, for future use.

7. Programming

This section gives some examples of how applications are programmed using the WIN system.

7.1 Example 1: Flashing Window

The code below implements a very simple application in which the interior of a window flashes whenever mouse button 1 is pressed with the cursor over the window.

```
! Assume windowCreator, screenWindowCreator, notifierCreator, windowManagerCreator, eventMonitorCreator
! and screen have already been read from the persistent store.

let flashingWindow = windowCreator( Dimensions( 100,100 ),Dimensions( 50,50 ),Dimensions( 125,150 ) )
! This window can't be resized to be smaller than 50 by 50 or larger than 125 by 150.

let flashApplication = proc( e : Event )
    ! First check whether the event is a mouse event, and if so whether button 1 is pressed.
    if e is mouse and
        e'mouse( buttons )( 1 )
    do ! Make the window flash by performing a not raster operation from itself onto itself.
        flashingWindow( windowRasterTo )( Limit( Pos( 0,0 ),Dimensions( 100,100 ) ),
                                          Limit( Pos( 0,0 ),Dimensions( 100,100 ) ),
                                          flashingWindow,
                                          Rule( Rnot : nil ) )

! Set the application for the window.
flashingWindow( setApplication )( flashApplication )

! Make a top-level window to operate on the screen.
let screenWindow = screenWindowCreator( screen )

! Make a notifier for the window manager to use.
let notifier = notifierCreator()

! Make a window manager and display the window at the top level; make it the current window.
let windowManager = windowManagerCreator( screenWindow,notifier )
let ok = windowManager( display )( flashingWindow,Pos( 50,70 ),Level( 1,true ) )
windowManager( makeCurrent )( flashingWindow )

! Start the event distribution system.
let finished = proc( → bool ) ; false
let eventMonitor = eventMonitorCreator( finished,notifier( distributeEvent ) )
eventMonitor()
```

This example has been slightly simplified: the display procedure in fact takes more parameters. These specify which border style should be used to display the window, and whether or not it should be displayed as a background window.

Note that this invocation of the WIN system will never terminate unless externally interrupted. Normally the result returned by the *finished* procedure depends on some system condition: for example, it might return the value of a boolean variable which was set to true when a **halt** light-button was pressed.

7.2 Example 2: Light-button Implementation

This example gives a possible implementation for light-buttons.

```
let lightButtonCreator = proc( buttonIm : Appearance ; buttonProc : proc( int ) -> InterfaceTool )
begin
      let buttonImage = if buttonIm is anImage then buttonIm'anImage else ! image with the string on it

      ! Create a window of the same size as buttonImage.
      let buttonDimensions = Dimensions( xDim( buttonImage ),yDim( buttonIm ) )
      let buttonWindow = windowCreator( buttonDimensions,buttonDimensions,buttonDimensions )

      ! Copy buttonIm onto buttonWindow.
      buttonWindow( imageRasterTo )( Limit( Pos( 0,0 ),buttonDimensions ),
                                     buttonImage,Rule( Rcopy : nil ) )

      ! Variable which shows which mouse button is currently pressed, if any.
      let lastButtonPressed := 0

      ! Inverts the button.
      let invertButton = proc()
        buttonWindow( imageRasterTo )( Limit( Pos( 0,0,0 ),buttonDimensions ),
                                       buttonImage,Rule( Rnot : nil ) )

      ! Create the application for buttonWindow.
      let buttonApplication = proc( event : Event )
        lastButtonPressed :=
          case true of
          event is mouse :
                if lastButtonPressed ~= 0
                then ! A mouse button was already pressed.
                      case true of
                      event'mouse( buttons,1 ) : 1
                      event'mouse( buttons,2 ) : 2
                      event'mouse( buttons,3 ) : 3
                      default : begin
                                      ! The mouse button has been released so restore the button to normal
                                      ! and call the button procedure.
                                      invertButton()
                                      buttonProc( lastButtonPressed )
                                      0
                                end
                else ! No mouse button was already pressed.
                      case true of
                      event'mouse( buttons,1 ) : { invertButton() ; 1 }
                      event'mouse( buttons,2 ) : { invertButton() ; 2 }
                      event'mouse( buttons,3 ) : { invertButton() ; 3 }
                      default : 0

          event is deselect : ! The cursor has moved off the button area.
                if lastButtonPressed ~= 0
                do  begin
                          ! unhighlight the button
                          invertButton()
                          0
                    end

          default : 0

      buttonWindow( setApplication )( buttonApplication )

      ! Return a structure with the light-button window and a procedure to make the button flash.
      InterfaceTool( buttonWindow,
                  any( proc() ; { invertButton() ; invertButton() } ) )

end
```

The definition of the type *Event* given in Section 2.1 was simplified. The full definition is as follows:

type Event **is variant**(chars : string ; mouse : Mouse ; select,deselect : null)

As well as events describing mouse and keyboard states, there are also two system-generated events, **select** and **deselect**. These are used to tell applications when they are about to begin or finish receiving a stream of events.

A notifier keeps a record of which of its notifications was the last to accept an input event (either a *chars* or *mouse* event). When the notification which accepts the current event is a different one from the last to do so, the notifier sends a deselect event to the application of the previous notification and then a select event to the new notification. This is done before sending the actual input event to the new notification.

This scheme allows applications to carry out some action when they first start receiving events after a period of inactivity, and also when they are about to become inactive again. This is illustrated in the light-button example above: when the application in the light-button window receives a deselect event, this means that the cursor has moved off the light-button, so that it no longer receives mouse events. If no select and deselect events were provided the application would not be able to de-highlight the button when the cursor moved away.

8. Implementation

This section discusses the implementation of the WIN system, in particular the way in which windows and window managers interact in controlling the storage of window information; the tree structure used by window managers to reduce image fragmentation; and the way in which input events are distributed to window applications and window borders.

8.1 Window Storage: General

There are two main approaches to window manager implementation. The first is to keep a complete copy of the contents of each window within the window itself. This results in relatively simple algorithms for window manipulation, as any part of a window can be recovered easily, regardless of whether it is covered by other windows. However, there is a storage overhead as the contents of the visible parts of windows are held both within the window and on the display screen. Another overhead arises from the fact that raster operations onto visible regions must be carried out twice: once onto the internal copy, and once onto the screen.

An alternative strategy involves keeping no extra copies of window information: the visible regions of windows are held on the screen only. This removes the store overheads and the double raster update problem, but the disadvantage is that the system's internal data structure and the algorithms to operate on it are correspondingly more complicated.

The predecessor to WIN, the PStools system, used the former method, partly because there was only a short period available to implement the system. It was felt that it would be worthwhile experimenting with a more sophisticated method for WIN.

8.2 Window Storage in WIN

The system used in WIN keeps no redundant copies of window information: the contents of any visible parts of windows are held in the screen memory and nowhere else. This involves some cooperation between windows and window managers. When a window is not displayed by a window manager, as is the case when it is first created, the contents are held as a single image by the window itself. When the window is given to a window manager for display, the window relinquishes control of its contents to the window manager.

This is implemented using **virtual windows**. When a window manager displays a window, it passes to the window a virtual window, which is an object with the same procedural interface as a normal window. Subsequently, when a raster procedure of the window is called, the window calls the corresponding procedure of the virtual window given to it, instead of performing the raster operation on its internal image (which it has discarded). The window manager constructs the virtual window so that its raster procedures operate on the window manager's own data structure rather than the single image previously held by the window. The window manager's data structure holds the contents of all the hidden parts of the windows displayed by it. When a window is undisplayed, the window manager gives it a new virtual window whose procedures operate on a reconstructed single image.

A window procedure, *setVirtualWindow : **proc**(Window)*, is provided for this purpose.

8.3 Window Manager Data Structure

The contents of the covered parts of windows are stored by the window manager in a doubly linked list, with one entry for each window displayed by the window manager. Within each entry a binary tree holds an image for each section of the window which is covered, and an alias onto the screen (in fact the higher-level window within which the window manager is running) for each visible section.

When a window becomes partially covered, it is divided into rectangular regions, each of which is either completely covered or completely visible. The way in which this division is carried out is reflected in the structure of the binary tree. This is illustrated in Figure 5.

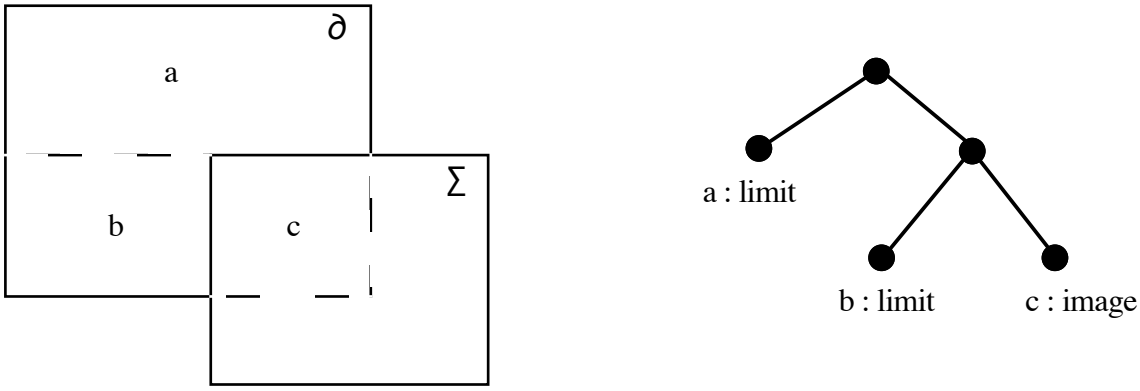

Figure 5. A Partly Covered Window (❶) and its Corresponding Binary Tree

In this example window ❶ is partially covered by window ❷ and has been split into three regions *a*, *b* and *c*. Part *a* is visible, and so the corresponding leaf in the tree contains a limit onto the screen on which the window manager is working. The remainder of the window is further divided into two parts, part *b* which is visible and part *c* which is obscured by window ❷. The leaf for region *b* also holds a limit onto the screen, while that for part *c* holds an image containing the hidden information.

The window manager also stores at each node the coordinates of the rectangle which that node represents. This holds for interior nodes (which do not themselves hold any window content information) as well as leaf nodes. This is so that tests for intersection can be carried out more quickly: if a point is not contained within the rectangle of a parent node then it is definitely not within any of the children's rectangles.

The window information is stored in a tree structure, rather than a simple linked list (as was used in the PStools system), in an effort to combat the problem of window fragmentation. This problem occurs when a window becomes divided up into many regions, as would happen in the situation illustrated in Figure 6:

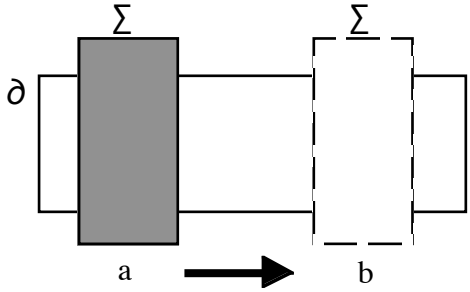

Figure 6(a). One Window Moved Gradually Over Another

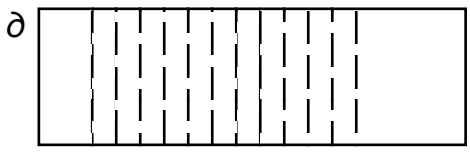

Figure 6(b). Resulting Window Fragmentation

If window ❷ is moved from position *a* to position *b* in a series of small steps, window ❶ behind it will become highly fragmented. This is because window ❶ is split further every time window ❷ is

moved to a new position above it. There are two problems associated with this fragmentation, the first of which is that the data structure describing the window's contents is much larger, and so window manipulation algorithms which have to traverse the structure will take longer. The other problem is that if window ❶ becomes completely hidden, e.g. if another window is moved on top of it, its contents will be stored as a large number of small images, and so the memory overheads associated with storing an image will become more significant.

To reduce these problems, a tree structure is used. A compaction algorithm, analogous to a garbage collector working over an object store, is invoked on certain occasions to recombine sibling leaf nodes where possible. It does this by recursively traversing the tree and examining each pair of leaf nodes. If both nodes are of the same type i.e. either both hidden regions or both visible regions, then the parent node is overwritten by a new leaf node constructed by combining the two leaf nodes. Figure 7 shows how some particular trees would be reduced by this algorithm:

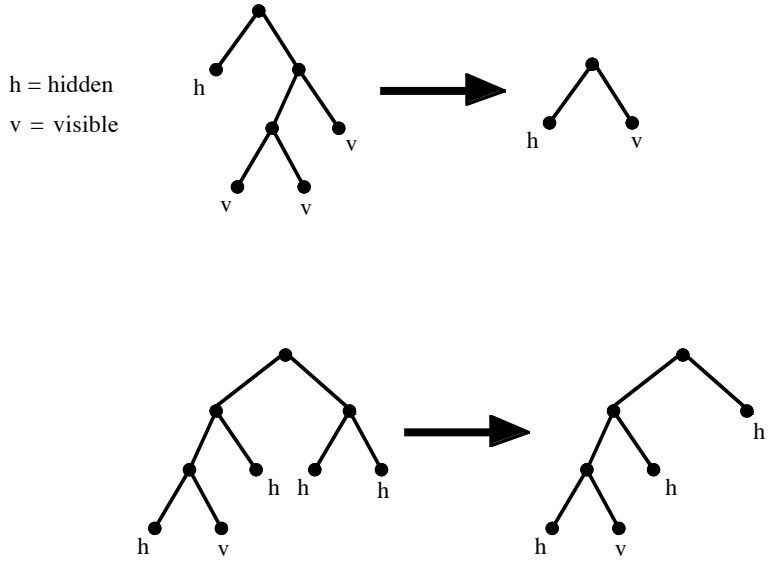

Figure 7. The Reduction of Some Trees by the Compaction Algorithm

This is a very simple algorithm: for example it could not perform any recombination for the tree in Figure 8:

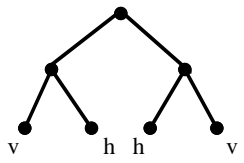

Figure 8. A Tree that will not be Compacted

However, the algorithm does appear to give significant improvements at low cost. It is invoked (for every window displayed) whenever any window is closed, opened, moved, displayed or undisplayed.

8.4 Border Implementation

When a window manager displays a window, it must display along with the main window area a border made up of a variable number of regions (the number depending on the border style used). These border regions may also be partially obscured by other windows or their borders. To cope with this the window manager maintains a separate binary tree for each region, so that for each window there is in fact a list of binary trees, of which the first entry is for the main area of the window, and the following entries correspond to the parts of the window's border. This implementation suffers from the disadvantage that the number of trees to be traversed when performing window manipulation is greatly increased. However, there was little choice given that a border may contain an arbitrary number of regions, and that the overall outline of the border need not be rectangular. This second factor precluded the solution used in the PStools system, namely holding one image with the border and main area combined. It was felt that it was better to build a flexible system and worry about speed later (in fact performance appears to be fairly satisfactory using this method).

To distribute input events to the applications associated with borders, a notifier hierarchy is used. Whenever a window is displayed, a new notifier is created for it. One notification is then registered with this notifier for each region making up the border of the window. The application part of the notification is the one in the *area* structure for that region. The *distributeEvent* procedure of the window's notifier is then used in constructing a notification for the window and its border as a whole, which is registered with the main notifier used by the window manager.

To speed up distribution of events to the current window (as normally the majority of events go there), the window manager maintains a separate notification for the current window at the top of the notification list. This has to be changed every time a new window becomes current.

9. Further Work

A number of developments to the WIN system are planned:

- When the Napier88 language is modified to support processes, the application for each window will be modeled by a process rather than a procedure. This will enable more than one application to be active at a time.
- Notifiers will be made polymorphic, being parameterised by the type of the events handled. This would generalise the system to cope with any kind of event rather than just the mouse and character events recognised currently.

- As well as background windows, the system could provide the option of displaying a window in the foreground, so that it would always be in front of all other windows. Such a window would be transparent, i.e. the windows behind would show through its blank areas. This would allow drawing over the top of the normal windows which is useful, for example, in implementing 'rubber-banding' during interactive window manipulation. This idea could be extended to allow windows at any level to be displayed transparently.
- A window-based text editor will be written.
- Experiments with the distribution of the system over a network will be conducted.

It is planned to use the WIN system to provide user interfaces during future experiments. These are likely to include developing integrated programming and debugging environments [12], and tools to support interactive browsing over a distributed persistent object store.

10. Conclusions

This paper has described a window management system in which the windows are treated as independent objects which can be manipulated by window managers or retained, with state intact, in a persistent store. It is believed that the tool reuse that this enables will give significant reductions in the cost of implementing applications.

11. Acknowledgements

We would like to thank Ron Morrison for his supervision and encouragement, and Richard Connor for his part in the designing of the event distribution system.

References

1. "Inside Macintosh",
Apple Computer Inc., Addison Wesley (1986).

2. "SunView™1 Programmer's Guide",
Sun Microsystems (1988).

3. Myers B.A.
"A Complete and Efficient Implementation of Covered Windows",
IEEE Computer, September 1986.

4. Pike R.
"Graphics in Overlapping Bitmap Layers",
ACM Transactions on Graphics, Vol. 2 No. 2, April 1983.

5. Morrison R., Brown A.L., Carrick R., Connor R.C.H. & Dearle A.
"Napier88 Reference Manual",…

6. Atkinson M.P., Bailey P., Cockshott W.P., Chisholm K.J. & Morrison R.
"Progress with Persistent Programming",
PPRR-8, Universities of Glasgow and St Andrews, Scotland (1984).

7. Atkinson M.P., Morrison R. & Pratten G.D.
"A Persistent Information Space Architecture",
PPRR-21, Universities of Glasgow and St Andrews, Scotland (1985).

8. "PS-algol Reference Manual"
PPRR-12, Universities of Glasgow and St Andrews, Scotland (1987).

9. Cutts Q.I. & Kirby G.N.C.
"An Event-Driven Software Architecture",
PPRR-48, Universities of Glasgow and St Andrews, Scotland (1987).

10. Cutts Q.I., Kirby G.N.C., Connor R.C.H., Dearle A. & Marlin C.D.
"An Object-Oriented Approach to Window-based Applications",
PPRR-72, Universities of Glasgow and St Andrews, Scotland (1989).

11. Cutts Q.I., Dearle A. & Kirby G.N.C.
"WIN Programmer's Manual", *in preparation*

12. Dearle A., Cutts Q.I. & Kirby G.N.C.
"Browsing, Grazing and Nibbling Persistent Data Structures",
PPRR-68, Universities of Glasgow and St Andrews, Scotland (1988).