CMU-ITC-86-045

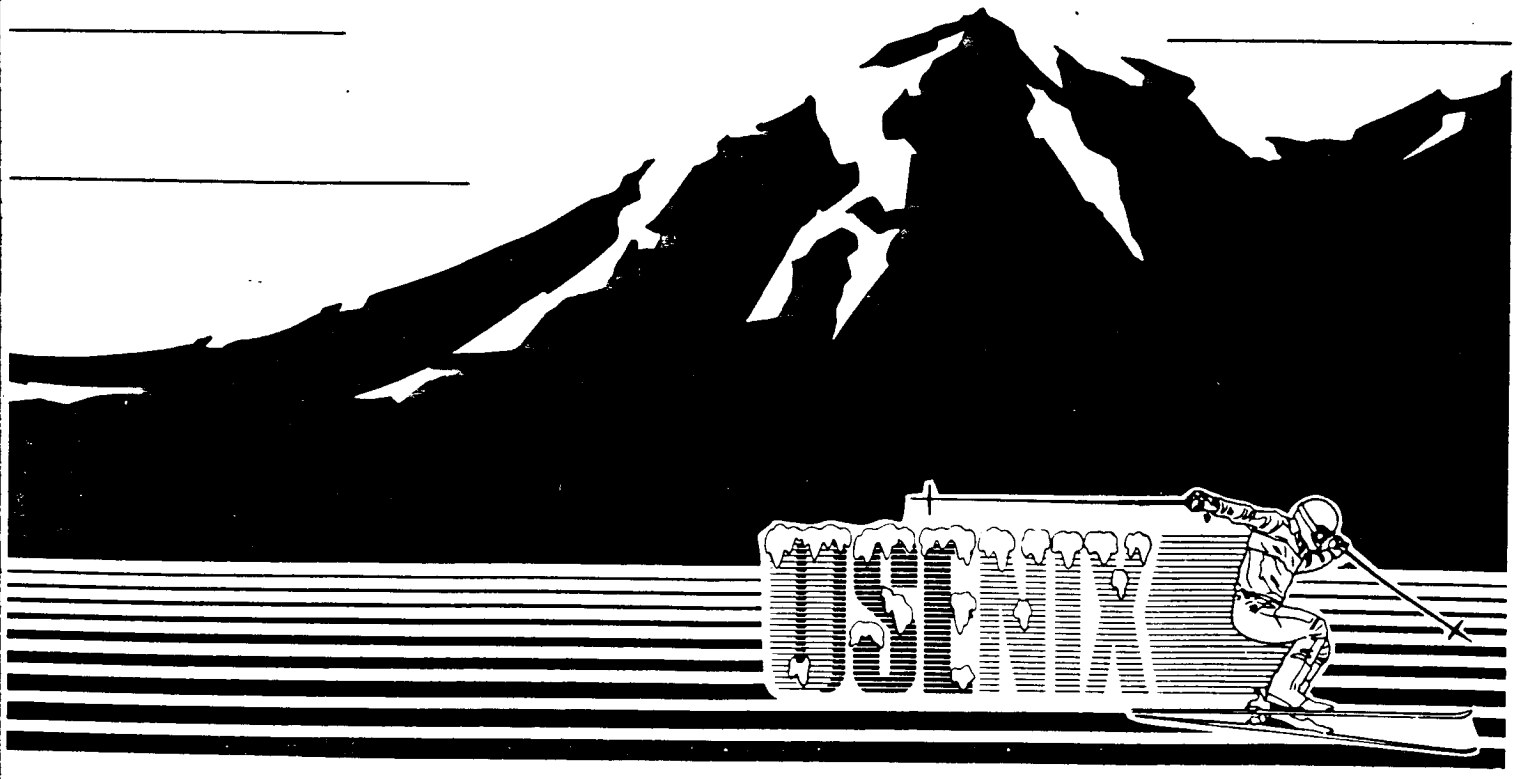
TUTORIAL MATERIALS

USENIX TECHNICAL CONFERENCE

Denver, Colorado January 15-17, 1986

#7

Windowing Systems Implementations



Window System Implementations

Denver Usenix Course Notes

David S. H. Rosenthal

Sun Microsystems
2550 Garcia Ave.
Mountain View CA 94043

ABSTRACT

Notes for a course given at the 1986 Winter Usenix meeting in Denver, CO. It covers window systems for Unix, primarily those for 4.2BSD, from the point of view of an application developer wishing to use one (or more).

1. Introduction

This course covers the implementation and facilities of window systems for 4BSD UNIX.† It is not intended for someone intending to implement a new window system, but rather for an application developer needing to understand the range of facilities available, the strengths and weaknesses of the current systems, and techniques that can be used to write programs which are easier to port between window systems.

1.1. To the Course

The course is organized roughly as follows:

- *Survey* – in which we examine some typical examples of the genre, for use as examples in the discussion. The systems in question are:
  - The Sun window system
  - Oriel, from Whitechapel Computer Works
  - Andrew, from Carnegie-Mellon University
  - X, from MIT.

  Others, such as the Blit, the LucasFilm system and those developed at Stanford will also be touched on.
- *Programmer's Model* – in which the surveyed systems are used to illustrate a generic model of a window system for Unix. The model covers three main areas:
  - Output
  - Input
  - User's Interface
- *Implementation* – in which a grab-bag of techniques that have proven useful in implementing window systems, and a range of common problems, are described.
- *Writing Portable Programs* – in which the diversity of programming interfaces among the systems is re-examined and some hints for survival in this world set out.

1.2. To the Author

This course represents a personal view of the range of window managers for 4BSD, and the opinions expressed are my own. To provide some insight into why I take these positions, here is a brief *curriculum vitae*:

I started computer graphics in 1968 on a PDP-7 with a DEC 340 display attached to Cambridge University's late lamented TITAN. I went on from there to do a Ph. D. on an application of hidden-line removal on a CDC 274 display driven via a CDC 1700 from the University of London's CDC 6600.

Next came research in the University of Edinburgh's Architecture Department, using Tektronix terminals driven initially from a PDP-10 and later (oh, joy!) from a PDP-11/60 running (hurrah!) UNIX Version 6-and-a-bit. Better graphics, more complex UNIX systems, and a VAX gradually came along to make our life better. However, I solved this problem by getting involved with the development of the GKS graphics standard, rising

† UNIX is a trademark of Bell Laboratories.

through the ranks to become the chairman of the British standards committee dealing with GKS.

I had also been working on window-manager style graphics systems using the Tek terminals, inspired by some film Alan Kay showed of Smalltalk in action. All these attempts really showed was that to do these things you needed better hardware so, when Sun started up and the Stichting Mathematisch Centrum in Amsterdam ordered some, I moved there for a year to write a window manager for them. A year later I had lots of good ideas, but no Suns!

I returned to Edinburgh for a while and then moved to Carnegie-Mellon. IBM had funded the Information Technology Center to spread high-powered workstations across the campus and at last I had found somewhere that really had machines (Suns) on which I could write a window manager. But the ITC also had someone else who thought it would be a good idea to write a window manager, so I got to kibitz while James Gosling wrote one. In effect, I was the user of the system for a while.

Two years later, James had moved to Sun to write the next generation of window managers. As it was starting to come together, he discovered a need for a user to test it, and so.....

1.3. To the Sources

The material for this course was assembled from a number of sources, and a bibliography will be provided. However, there are a few sources that deserve special attention:

- In December 1984 Usenix sponsored a workshop on Unix and Graphics at Monterey, CA. It led to some fruitful discussions but, unfortunately, the proceedings were never published. A second workshop was held in December 1985 (after the copy deadline for these notes), and these proceedings are available from Usenix.
- In April 1985 the Alvey Directorate, responsible for the British "5th Generation" project, sponsored a useful workshop on window managers at Abingdon, near Oxford. The proceedings of this workshop are being published by North-Holland. Special thanks are due to the Rutherford Laboratory, for hosting the workshop, David Duce, for recording the proceedings in readable form, and among the attendees to Dominic Sweetman of Whitechapel Computer Works, Colin Prosser of ICL, and Warren Teitelman of Sun Microsystems for volunteering their hardware and software as sacrificial victims.
- During the development of the Andrew window system at C-MU, a vast amount of feedback and many design suggestions were received from the application developers trying to use it. Special thanks are due to the ITC's user interface group, James Gosling (who built the original server and editor), Fred Hansen, Bruce Lucas, Andy Palay, Thom Peters, and Jim Peterson, to the Center for the Design of Educational Computing (Bruce Sherwood, Tom Neundorffer, and Chris Koenigsberg), to Sandra Bond of the Communication Design Center, to Dan Boyarski and his graphic design students, and to the numerous groups on campus who are building educational software in the Andrew environment.
- During the development of MIT's X window manager, a series of discussions was held between teams from MIT's Project Athena and the ITC. Special thanks are due to Jim Gettys of Project Athena, who participated in them all. The useful results are

(I hope) visible in the design of X.

2. Survey

Window systems for UNIX have been implemented in two very different ways.

- Some give client applications the impression that the pixels on the screen are in their address space, and can be manipulated directly. They use major kernel extensions, and typically link large libraries of graphics operations into every client process. I call these systems **kernel-based**.
- Some use the 4.2BSD socket mechanism to implement a network window service, permitting clients to use remote procedure calls (RPCs) to operate on pixels which are clearly not in their address space. I call these systems **window servers**.

The following survey covers two of each type in some detail, identifying the components that are supplied, the facilities implemented by the library or server, the extent of any kernel support required, and the portability of the system. We start with the kernel-based systems.

2.1. SunWindows

SunWindows has been developed by Sun Microsystems to support their range of 4.2BSD-based workstations. The machines use the MC68010 and MC68020 processors and a range of displays, some of which appear to be memory, and some of which have various kinds of RasterOp support hardware. None have cursor support hardware.

The system was the first widely used window system for 4.2BSD. Among kernel-based systems it is the most mature, and it is still evolving. This description refers primarily to Release 2.0; Release 3.0 is in beta-test, it includes significant extensions to the upper layers of the system, and several new clients. Steve Evans was very helpful in correcting some of my misunderstandings of the system.

2.1.1. Components of the SunWindows System

SunWindows consists of a library, a set of client programs built using the library, a daemon program, and a set of kernel modifications to support them. The system is implemented as a set of layers:

- Windows - a hierarchy of overlapping rectangular regions of the screen, each capable of manipulation as a unit. Each display is represented by a single root window.
- Sub-windows - a non-overlapping division of windows into areas that can be drawn in and that can supply input events.
- Pixwins - overlapping rectangles with associated locking and clipping. The basic level at which shared screen access is implemented.
- Pixrects - a device-independent representation of bitmaps and the operations on them.

A number of other components fit into this structure:

- Windows, subwindows, and Pixwins are implemented using extensive kernel support. The kernel maintains the window placement tree, arbitrates access to shared resources, and routes input to the `/dev/win*` files corresponding to the window with the input focus.
- The user interface to window manipulation uses special cursor icons, pop-up menus and the mouse. It is implemented using subwindow-level code that is linked into

every client.

- The user enters the SunWindows environment by running a program called `sun-tools`. This program grabs the keyboard and the mouse, enables the kernel support, and owns the root windows, painting them as requested in desktop grey.

2.1.1.1. The Libraries

The libraries implementing the various levels of the layered structure provide the following facilities:

- Client programs operate in **windows**, which are rectangular regions that may overlap on the screen. Each window corresponds to a device file `/dev/win*`. Mouse and keyboard events are demultiplexed by the kernel and routed to the appropriate one of these devices. When windows change size, or become exposed, the client is notified and must re-paint the image.
- Clients may divide their windows into **subwindows**, which tile the window. They are also associated with device files, and behave in other ways just like windows[†]. Subwindows may be occupied by a library of pre-defined **panels**, implementing behaviours such as text panes, button arrays, and so on. Subwindows may specify the shape of the cursor when it is inside them. Input events may be received from subwindows., and they may implement pop-up menus of various kinds.
- The output aspects of subwindows are implemented using **Pixwins**; structures encapsulating the clipping and locking necessary to arbitrate access to overlapping rectangles on a display shared between multiple processes.
- Pixwins are implemented using **Pixrects**; structures encapsulating the storage of rectangular bitmaps together with the implementations of suitable RasterOps to paint in them.

2.1.1.2. The Clients

As it has evolved, SunWindows has acquired a large number of clients. The most important Sun-supplied ones include:

- `shelltool` – a terminal emulator window.
- `gfxtool` – a program that establishes a window and runs a program that takes it over as a window to do graphics in.
- `tektool` – emulates a Tektronix 4014 terminal for backwards compatibility.
- `dbxtool` – a panel-based interface to the `dbx` debugger, interacting in one panel and displaying the program text in another.
- `clocktool` – the canonical window program.
- `perfmeter` & `perfmon` – two different styles for displaying system performance parameters.
- `icontool` – a bitmap editor, useful for creating the icons that represent closed windows.

---

† - In fact, they are implemented using the same kernel support as windows. The restriction that they not overlap is imposed by the user-level libraries.

- `fonttool` – an editor for fonts in the `vfont` format.

The 3.0 release includes a number of new tools based on scrolling text panels, including a shell interface and a text editor. Tools based on these panels implement inter-window cut-&-paste (implemented using a special daemon called the selection service).

2.1.1.3. The Utilities

SunWindows provides many utilities, for manipulating raster image files, desktop specifications, and so on. Two are worthy of special note:

- `adjacentscreens` – allows a user to specify the relative location of multiple displays, so that the mouse will track smoothly across their boundaries.
- `lockscreen` – a program that keeps the screen dark and unusable until the password owner returns.

2.1.1.4. The SunTools program

In SunWindows, the root window on each display is just a window like any other, and it too must be repainted when it is damaged. The program that does this is called `suntools`; it is the command the user executes to enter the SunWindows environment. It has several functions:

- It initializes the window environment by grabbing the mouse and the keyboard and enabling the kernel support.
- It starts up the user's default windows according to specifications in a profile file.
- It repaints the desktop grey background when it gets damaged.
- It implements the menu that pops-up over the desktop. This provides for starting new instances of the common clients, for re-painting the entire display, and for shutting down the environment gracefully.

2.1.2. Facilities of the SunWindows Library

The SunWindows libraries are intended to support three rather different areas:

- A device-independent interface to bitmap operations.
- Windows and their manipulation.
- A user-interface toolkit of panels of various types.

2.1.2.1. Device Independence

The device-independent interface to bitmap operations is the Pixrect:

```
struct pixrect {
        struct    pixrectops *pr_ops;   /* operations appropriate to this pr */
        struct    pr_size pr_size;      /* pixels per dimension */
        int       pr_depth;             /* bits per pixel */
        caddr_t   pr_data;              /* device-dependent pixel-access */
};
```

It describes a stored bitmap, and points to the set of operations appropriate to manipulate it. The operations include:

- Create and destroy Pixrects.
- Read and write individual pixels.
- RasterOp, with or without a mask bitmap.
- Replicate a source pattern.
- Draw a straight line and a polyline.
- Draw text in a selected font (but with no shimming).
- Fill an area bounded by a polygon or a **trapezon**; a trapezoid-like figure whose non-parallel sides are specified by chains that can describe curve approximations.
- Make a new Pixrect representing part of an existing one.
- Manipulate the color map for a Pixrect.

Note that the encapsulation of both the storage of and the operations on a bitmap allows the Pixrect level to present a uniform interface to higher-level software that supports both monochrome and color displays.

2.1.2.2. Windows

The window support in the library can be divided into three main areas:

- Pixwins, an extension of Pixrects to cover the requirements of overlapping and sharing between multiple processes.
- A user interface to window manipulation.
- A device-independent interface to input events, both from the mouse and keyboard, and synthetic events.

Windows at this level are represented by `/dev/win*` files and many of the operations involve special semantics for the `read()`, `write()`, and `ioctl()` system calls on these files.

2.1.2.2.1. Pixwins

To support overlapping and multi-process access to Pixrects, the Pixwin structure adds to the basic pixrect structure:

- pointers to the shared objects (the display device and the color maps),
- an optional pointer to a pixrect used to retain a pristine copy of the bitmap,
- and a set of operations to lock, unlock, and get the clip list.

Before drawing in a Pixwin, a client must lock it. The lock call may block, but when it returns it supplies the clip list and guarantees not to make changes to the display that would invalidate the list until the client unlocks. A client observing the locking protocol and the clip lists will behave correctly even though it may be overlain by other windows, and may be sharing the color map with them.

2.1.2.2.2. Window Manipulation

Linked into every client is code implementing the user interface to window manipulations. A window can be open, or closed into an icon. Windows may be moved, resized, pushed to the bottom of the window stack, and popped to the top. All these operations are available from menus that pop up wherever the client has not specified some other

menu. By convention, clients draw a border around their active areas, and do not specify their menus there.

2.1.2.2.3. Input

Just as there are problems with different display devices, there are problems with different keyboards, mice, tablets, and so on. SunWindows supports a wide range of input devices in two ways:

- By virtualizing the input devices so that they all generate events. An event is represented by a structure:

```
typedef    struct     inputevent {
           short      ie_code;              /* input code */
           short      ie_flags;
           short      ie_shiftmask;                    /* input code shift state */
           short      ie_locx, ie_locy;     /* locator (usually a mouse) position */
           struct     timeval ie_time;      /* time of event */
} Event;
```

  The code defines the type of event, such as an ASCII keypress, a mouse button transition, a mouse motion, or a synthetic event such as a window boundary crossing. The rest of the event contains the state of the shift, meta and function keys, the location of the mouse, and the system time at the occurrence of the event. Note that this structure allows for many styles of input handling, including multi-clicking, modifying the effect of mouse clicks by holding down function keys, and so on.

- by serializing them into a single stream and distributing events from this stream to the appropriate window. Clients read event structures from their `/dev/win*` file descriptors; the particular file an event gets sent to is normally determined by the window the mouse is in at the time. In fact, there are two separate *Input Foci*, one directing keystrokes and the other mouse and menu events.

  Clients are expected to field window crossing events, and paint their boundaries and carets to indicate whether they have the input focus. This poses a performance problem; moving the mouse between two windows requires the process with the old window to be scheduled to show that it has lost the focus, and then the process with the new window to be scheduled to show that it has acquired it.

2.1.2.3. Panels

The user interface toolkit in the SunWindows library allows applications to be assembled rapidly from pre-defined panels and pop-up menus. It consists of:

- a core program that maintains a dynamic structure of panels
- a defined interface between the core and panels
- a library of pre-defined panels.

The core fields re-paint requests and calls routines defined by each panel to perform them, and reads all input events, calling the routines nominated by the panels to receive the different event types.

2.1.2.3.1. Panel types

Pre-defined or application-specific panels may be created and composed according to a number of geometric attributes. They may be just wide or high enough to fit their contents, to fill the window with a specified margin, to occupy a specified proportion of the window, and so on. The panel types supported include:

- message – displaying a text string
- button – displaying a click-able button whose image may be a string or an icon.
- choice – displaying a menu of click-able buttons whose images include a selection feedback.
- slider – displaying a bar with a pointer that may be dragged between a low and a high value.
- text – displaying text that may be edited.

2.1.2.3.2. Menus

SunWindows supports menus that pop-up on the right mouse button in a number of styles, including a hierarchical pull-right style. The images of items in a menu may be graphic or text. The menus that pop-up are a property of the window the mouse is in when the button is pressed.

2.1.3. Kernel Support Required by the SunWindows Library

The kernel support for SunWindows is extensive and somewhat complex. In outline, the kernel maintains a separate hierarchy of **panes** (Sun calls them windows, but this is somewhat confusing) for each display. At each level of the hierarchy, panes overlay their older siblings. Child panes overlay their parents, but are clipped to their parent's outline. The root pane represents the display surface. Panes may be detached from the hierarchy, in which case they are invisible, and may be inserted in arbitrary positions in the hierarchy. System calls are provided to:

- Interlock between multiple processes attempting to access shared data. The shared data includes the pane hierarchy, the color map, and the pixels on the display surface. *Note that the integrity of the display is entirely dependent upon clients locking around their accesses to these resources; there is no mechanism for enforcing locking.* The care taken by each client to determine a suitable granularity for these lock operations is important to overall system performance. Locking at a fine granularity, for example round each RasterOp, results in excessive system call overhead. Locking at a coarse granularity results in other processes waiting unreasonable lengths of time to access the display. Locks held too long are broken by a time-out, safeguarding the system from completely anti-social clients.
- Create & destroy panes by opening and closing their associated `/dev/win*` files.
- Insert and remove panes from the hierarchy.
- Associate cursor images with panes.
- Read and mask events from the `/dev/win*` file. The window system multiplexes keyboard and mouse events into a single stream of events, and distributes them to the pane holding the keyboard focus or containing the mouse.

A newly inserted pane must be painted. It may be (partially) overlapped by other panes, and so must be clipped to a set of rectangles. A system call is provided to enable a process to discover the set of clip rectangles it must use. *Note that the integrity of the display is entirely dependent upon clients observing these clip lists; there is no mechanism for enforcing them.*

As panes are removed and inserted in the hierarchy, the overlapping relationships change, and parts of underlying panes become visible. The processes owning these panes are notified using a special signal (SIGWINCH), and must then use a special system call to discover the part of their image that needs re-painting. This mechanism is also used to re-size panes. *Note that the integrity of the display is entirely dependent on the clients responding to SIGWINCH, inquiring the extent of the damage and the new clip list, and re-painting their image.*

The SunWindows libraries are rather large, and it is also necessary to link complete device drivers for all possible Sun displays, and all the window manipulation code, into each client program. This results in a very large process (400+K of text) behind each window. Users will want many windows, and will thus run out of swap space, quite apart from the performance problems of paging the same code in over and over again. In the absence of shared libraries, Sun has attacked this problem by linking most applications into a single object file that acts (for example) as the shell tool if it is called `shelltool` and as the debugging tool if it is called `dbxtool`. The text for all instances of this merged tool is shared, achieving some of the benefits of shared libraries in reducing swap space and paging load. Of course, this technique is generally applicable; it requires no special kernel support

2.1.4. Porting the SunWindows System

The SunWindows system has not been ported to other workstation architectures, but has been ported to a number of different displays. Doing so involves creating a *Pixrect Driver* for the display. The Pixrect level of the system defines a common data structure to represent bitmaps, and a set of operations for manipulating them. Many Sun displays appear just like memory, and may use a pre-defined memory RasterOp to implement these operations; other displays may require custom code for some or all operations. In principle, this interface hides the display-dependent details such as byte and pixel order.

The new Pixrect driver must be linked into all applications that may use the device, and at least some minimal version of it must be linked into the kernel to support cursor tracking.

The only other parts of the system that are significantly machine-dependent are the low-level mouse and keyboard drivers. As none of the Sun displays have hardware cursor assistance, the mouse drivers have to paint the cursor into the bitmap themselves, and perform the necessary interlocking with the client's RasterOps. This is a major overhead, but exploiting cursor hardware would need significant changes to the drivers.

2.2. Oriel

The Oriel window system has been developed by Whitechapel Computer Works, a British workstation company. It is supplied as part of the 4BSD-based operating system on their low-cost NS32016-based MG-1 workstation. The machine has a number of

interesting features, in particular the display is refreshed from main memory via a page map, and a Z-80 processor off-loads the CPU by providing autonomous mouse-tracking with a video-mixed cursor.

The system is of interest for this course because it represents a library-based window system with a more coherent structure than SunWindows, and a very different type of kernel support. What follows is a summary of the information in the Release 2.0A manuals for Oriel, kindly supplied for this course by Dominic Sweetman of WCW, and in his presentation to the Alvey workshop.

2.2.1. Components of the Oriel System

The Oriel system supplies a number of components. They include a library of graphics and input operations, a number of client and utility programs built using this library, a window-manager daemon process that provides a user interface to window manipulation operations, and a kernel driver (panelist) that supports them all.

The system supports two key concepts. Applications operate in terms of **panels**, rectangular regions that may become visible. An application obtains one or more root panels from the window manager daemon, and can subdivide them into child panels. It can pass ownership of panels to other processes, attach bitmaps to them, perform graphics operations on the bitmaps, and obtain input events from them.

Users operate in terms of **windows**, overlapping rectangular regions of the display with or without a border containing **controls**. When an application obtains a root panel, the daemon will decorate its border with other panels. It owns these panels, and they form a standard set of buttons. The corner areas of the border are identical for all windows; they are used to signal the daemon to move, re-size, and pop or push the window in the stack. The remaining border areas are used at the application's option for scroll and split controls. The total set of panels is manipulated as a whole, and forms a **window**. In this way the user sees a uniform Mac-like interface to window manipulations; they are all performed by the daemon, which has resource names for and creator's rights to all root and control panels.

2.2.1.1. The Library

The Oriel library provides functions in a number of groups:

- Graphics functions in bitmaps, including RasterOp, text and line-drawing.
- The Panelist interface, permitting panels to be divided into child panels, bitmaps to be assigned to panels, input events obtained from panels, and panels to be destroyed.
- The interface to the Window Manager Daemon, providing for processes to request the creation of windows, and to obtain the corresponding root panels. This interface also provides operations on windows and icons such as push, pop and resize, specifically for use by the window manager daemon itself.
- The Toolkits, encapsulating the common uses of the facilities provided in the previous sections in easy-to-use form:
    - Basic Tools, encapsulating the operations required to create graphics and TTY emulator windows, endow them with icons and other attributes, manage their event queues, and deal with their interactions with the minutiae of UNIX, such as

the signal mechanism.

- Border Styles, implementing a standard form for the optional controls on windows. These are vertical and horizontal scroll bars, and controls for creating, destroying and moving splits. Applications using these border styles get an extended set of input events including "window scroll up" and "horizontal split moved".
- Menu Package, implementing a single-level pop-up menu style, with menu items as either text or icons.

2.2.1.2. The Clients

The Oriel system provides a fairly limited number of clients:

- `newwin` – a TTY window interface to the shell and other UNIX commands. It uses the panelist's built-in VT100 emulator to support `termcap` programs such as `vi`.
- `console` – a TTY window interface to `/dev/console` messages.
- `draw` – a MacPaint subset.
- `iced` – icon and cursor bitmap editor
- clock
- `calc` – a pocket calculator
- `petal, saddle` – spirograph windows

2.2.1.3. The Utilities

The utilities mostly manipulate the desktop:

- `desksave` – save the current desktop layout for later re-creation.
- `pattern` – change the desktop pattern.
- `prtsc` – dump screen for Epson printer.
- `show` – temporarily pop all icons to the top, to help find the one you want.
- `tidy` – arrange the icons neatly.

2.2.1.4. The Daemon

- `wmgr` – A daemon process that manages the desktop, responding to user interactions to allocate screen space and to process' requests for new panels.

2.2.2. Facilities of the Oriel Library

The Oriel library duplicates some facilities available in 4.2BSD, because it was originally implemented under 4.1. For example, it uses PTYs as an IPC mechanism.

2.2.2.1. Graphics functions

Oriel's graphics functions are performed on bitmaps, storage structures private to user processes. For them to become visible, the raster must be attached to a panel, and the panelist notified that the panel has changed. It will then copy the changed parts to the screen. The operations supplied include:

- RasterOp, with the source optionally being a tile that is replicated as required to fill the destination with pattern.
- Line, Arc, and Circle, all single-pixel wide.
- Fill a region of contiguous black or white pixels with a pattern.
- Text in a specified font. No support is provided for shimming justified text; it would have to be painted a character at a time.

2.2.2.2. The Panelist interface

To make bitmaps visible, they must attached to panels. The Panelist interface functions provide for:

- dividing panels into child panels.
- attaching bitmaps to panels.
- destroying panels.
- obtaining input events from panels.
- dividing panels into regions with associated cursors.
- attaching the VT100 emulator to a panel (thereby creating a new TTY-like device `/dev/ttyv*` capable of supporting `termcap` style applications).

2.2.2.3. The Interface to the Window Manager Daemon

The window manager daemon performs screen space allocation. It owns a panel representing the entire physical display, creates sub-panels in response to requests from processes requiring new windows, and allocates them screen space in response to requests from the user. It implements the user's interface to operations such as hide, expose, and re-shape windows. In principle, different daemons could be provided to implement different user interfaces and window layout policies. The daemon interface provides functions to:

- Create & Destroy windows. Processes build trees of panels to form their displays. Actual screen space can be allocated only to the root of such a tree. To obtain a root panel, a process performs remote procedure calls on the window manager daemon specifying the maximum size, initial position, etc.
- Change window ownership. Windows are named by handles, small integers returned by `WindowCreate()`. The name space is global, so any process may name any window. To perform operations that change a window's state, a process must be either the creator or the owner. Typically, a window is created by the window daemon, which then hands-off ownership to the process requesting the creation. Processes can also transfer ownership of windows to their children.
- Pop & Push windows. Windows are arranged in a stack on the desktop, and the window manager can manipulate this stack.
- Re-size windows. When creating a window, a process specifies the maximum size it will attain. This information is required so that the necessary bitmaps can be allocated. The window manager interacts with the user to assign an actual size and position. These actual sizes can be changed by user actions; the process can inquire the current values, and ask to be presented with events when changes occur.

- Move and show icons. A process requesting a window can associate with it an **icon**. >From the process's point of view this is just another panel, but the window manager treats it differently. It isn't decorated with controls, and it is included in the set of panels to which icon operations apply.
- Stow and unstow windows from icons. The window manager can map (make visible) and un-map the collection of panels forming a window. It typically does so in response to clicks on the associated icon. Icons remain visible on the desktop irrespective of whether their windows are visible.

2.2.2.4. The Toolkits

2.2.2.4.1. Basic Tools

- The basic toolkit provides the common sequences of operations that applications use to access windows (in the looser general sense):
- Create and destroy windows, both graphical and VT100 emulators.
- Set and inquire window attributes, such as title strings.
- Manage windows across `fork()`s.
- Manage the interaction of windows with the UNIX signal mechanism.
- Manage the window's event queue, and obtain events from it.

2.2.2.4.2. Border Styles

- The border styles toolkit implements a standard form for the optional controls on windows. They include vertical and horizontal scroll bars, and controls for creating, destroying and moving splits. Applications using these border styles get an extended set of input events including ''window scroll up'' and ''horizontal split moved''.
- Create & destroy controls on a window.
- Create & move splits in windows
- Set range of scroll bar.

2.2.2.4.3. Menu Package

The menu package implements a single-level pop-up menu style, with menu items as either text or icons:

- Create & destroy menus.
- Set the image of menu items, as text or icons.
- Enable & disable menu items. Disable items are displayed grayed-out.
- Mark menu items (e.g. as selected) by a character (normally a tick) to the left.
- Display & hide menus. Menus are displayed at the mouse, or another specified position. They can be left displayed while others pop up, to give the visual impression of a multi-level menu scheme.

2.2.3. Kernel Support Required by the Oriel Library

The kernel support (panelist) for the Oriel system manages a hierarchy of **panels** each associated with a **page** in a user process; a panel-size bitmap containing the process' view of its contents. >From time to time the process notifies the kernel that part or all of the contents have changed, and the kernel copies the changed parts to the bitmap it maintains, from which the display is refreshed. Note that this implies that the kernel needs an implementation of RasterOp, albeit only a Copy operation.

Since the **page** is in a user process, it may get paged or swapped, and thus be unavailable when needed to re-paint an uncovered part of the display. To avoid lengthy delays in re-painting, the panelist maintains complete off-screen copies of obscured panels in kernel memory. Thus updating a (partially) obscured panel is twice as expensive as updating a panel on top. *Note that a client has no responsibility for observing clipping restrictions or damage lists, and need never re-paint its display.*

A panel can be subdivided into regions, and a cursor image associated with each. The panelist communicates these rectangle sets and associated images to the autonomous mouse tracking hardware.

The panelist maintains an event queue for each panel, with an associated mask. Keyboard transitions, mouse button transitions, and mouse movements are all formatted into an event report structure and added to the appropriate event queue. The owner of a panel can read events from this queue. The panelist also synthesizes events in this stream for particular windows to inform them that they have been made visible or invisible, or have changed size. *Note that is is not essential that clients respond to these events.*

The panelist protects panels, restricting state-changing operations (update, writes to attributes, move in hierarchy) to the creator and owner processes.

The panelist also contains a VT100 emulator, which can be attached to a panel to display its screen, and which generates a TTY-like device in the file system that can be used by `termcap` applications.

2.2.4. Porting the Oriel System

The Oriel system is being ported to the Sabre, which also uses the NS32016 chip, but whose display is not bitmapped, using instead a display-list type interface. No other attempts to port the system are known. The panelist would divide into parts that were display-dependent, CPU-dependent, and machine-independent.

- Machine-independent parts would presumably include management of the panel hierarchy, scheduling of copies from user pages and kernel buffers, buffer maintenance, event reporting, VT100 emulation.
- Display-dependent parts would include allocation of video refresh buffers, and the copy RasterOp (the display has hardware RasterOp support).
- CPU and I/O device dependent parts would include the interfaces to the mouse, keyboard, and cursor support.

Given the panelist support, the library would be display-independent. The RasterOp code would be dependent both on the CPU and the RasterOp hardware; it isn't easy to write a fast software RasterOp for the NS32016, and there are complex trade-offs in

deciding the point at which the investment in setting up the RasterOp hardware pays off.

2.3. Other Kernel-Based Systems

- The window system developed at LucasFilm by Sam Leffler and Mike Hawley, and described at the Portland Usenix, is another example of a kernel-based system. It supports an impressive user interface toolkit, similar to that on the Blit.

2.4. Andrew

The Andrew window system was developed at the Information Technology Center at Carnegie-Mellon University. The ITC is funded by IBM with a mission to develop a campus-wide network of personal workstations. The system is owned by IBM, but it is made available to Universities and some others.

Andrew provides a network window server, and a set of client programs that make remote procedure calls to the server to perform graphics operations and receive input. It uses 4.2BSD TCP/IP streams and a special low-overhead "batching" RPC mechanism.

The Andrew server retains no information about the contents of windows, and must call upon the clients to re-paint them when they are re-sized. The server lays non-overlapping windows out in columns, thereby reducing the frequency of re-sizing events.

2.4.1. Components of the Andrew System

2.4.1.1. The Server

The Andrew server supports the following configurations:

- Sun100 monochrome and color.
- Sun/2 monochrome and /160 color.
- Sun/3 monochrome and /160 color.
- micro-VAX with QVSS display.
- Experimental IBM workstation.

The server implements graphic output operations in windows, a non-overlapping window layout policy with a very simple user interface, and a menu system. The user-level server code tracks the mouse with a cursor; the shape of the cursor is an attribute of rectangular sub-divisions of windows called regions.

The design goals for the server were to implement a simple, highly portable, low investment system as a basis for development of the overlying user interface toolkit. The intention of both the toolkit and the server is to enforce a uniform consistent user interface on the educational applications that are the intended clients.

2.4.1.2. The User Interface Toolkit

Andrew's User Interface toolkit consists of:

- a core that maintains a hierarchical division of a window into panels, and distributes input events to panels,

- a set of behaviours for panels, including scroll bars, button arrays, large scrolling menus, meters, and other kinds of objects.

An application specifies its display as a set of panels linked by relationships such as `above()` and `left()`. Some of the panels may be defined especially for the application and others loaded from the toolkit.

The most important panel is the text panel, which implements a WYSIWYG-style editor supporting dynamic reformatting of multi-font proportionally-spaced kerned text at every keystroke. It operates on text files divided into regions with associated looks to control the formatting. Among the attributes a look can control are the font family, the face type (roman, bold, italic, bold italic), the size, the margin settings, the tab settings, and the justification modes. The result is similar to on-the-fly `Scribe`.

Text panels are used to display almost all text in the system; the 24-by-80 terminal emulator is the only significant exception. They are normally used in a composite form called a **document**, consisting of a text panel with a scroll bar to the left and another small text panel below acting as a message line. Text can be cut and pasted between documents using the server's cut-buffer facilities; if appropriate it retains its looks (i.e. if some bold text is cut it will be bold when pasted). Documents can read into panels and output to files in a special internal form, as `troff` input, or as `Scribe` input. The `troff` and `Scribe` forms are used to print documents.

2.4.1.3. The Clients

Andrew provides a large number of clients, all of which are known to run on Suns, VAXen and "experimental IBM workstations". Some, typically the basic tools, have been developed at the ITC, some at the Center for the Design of Educational Computing (CDEC), and many by application developers on the C-MU campus, who were given workstations as part of a major deployment in early 1985.

2.4.1.3.1. Basic Tools

The basic tools provide the interface to the normal UNIX facilities. They are mostly built from documents.

- `edittext` – an editor built from a single document panel. It provides all the capabilities of documents as described above.
- `typescript` – a single document panel providing an interface to the shell or other commands. Text that you type is bold, and text the computer types is roman, though the font used is up to the user. Text can be cut and pasted, and the document can be printed, just as any other document.
- `pipescript` – a single document panel into which text can be piped. Useful for the usual:

  ```
  grep foobar | pipescript
  ```

  sort of things.
- `ttyscript` – a single document on the master side of a pseudo-tty. Useful for the usual:

  ```
  interactive-program <`ttyscript`
  ```

sort of things.

- `h19` – emulates a Heathkit H19 terminal with great precision. Chooses a fixed-width font of an appropriate size so that the 24-by-80 array as nearly as possible fills the screen. Implements cut-and-paste. Used only for backwards-compatibility with `termcap` applications, such as `vi`, and `telnet` to other machines.
- `help` – a single document presenting the manual pages properly formatted to the window, with font and size changes, and cut-and-paste.
- `ringmaster` – a single document making the contents of the ring of cut-buffers visible (and thus editable and even cut-and-paste-able!).
- `male` & `female` – a pair of multi-panel applications using large scrollable menus and button arrays to select mail, and documents to display and edit it. It is possible to mail text containing the full set of looks, and have it displayed properly at the recipient's machine.
- `hark` & `bark` – a similar pair of tools for the Usenet.

2.4.1.3.2. File System Browsers

Many file system browsers have been developed as experiments towards a user interface usable by freshmen. They include:

- `dir` – a file system browser using large scrollable menus of files and pop-up menus of operations.
- `bush` – a file system browser using a graphical presentation of the hierarchy and a menu panel.
- `Don Z.` – an iconic file system browser. Icons represent both files and operations; file icons are dropped on operation icons.

2.4.1.3.3. Diagram Editors

Many diagram editors have been developed, for no very obvious reason. They include:

- `de` – the first, developed by Marc Donner. It uses pop-up menus of operations and attributes.
- `banzai` – a constraint-based drawing editor similar to Juno, developed by Bruce Lucas. It uses an iconic menu to access the underlying constraint programming language.
- `fig` – was developed for the SunWindows system at the University of Texas, and was ported to Andrew by Jim Peterson.

2.4.1.3.4. Font Tools

Andrew makes extensive use of fonts. It is normally used at C-MU with a set of high-quality display fonts owned by Xerox, but these cannot be distributed. The system supplied outside contains a large collection of public-domain fonts, in sizes from 6 to 36 points, in roman, bold, italic and bold-italic faces, and in serif, sans-serif, and typewriter styles. They were derived from Metafont descriptions at 240 dots/inch and reduced to 80 dots/inch using an adaptive resolution-reduction process. The results are somewhat scruffy but usable. The font manipulation tools include:

- `fe` – a font editor, permitting control over all the spacing and size parameters of individual characters, and its bitmap representation.
- `samplefont` – presents sample text in a given font as an aid to editing.
- `viewfonts` – a browser for the font library.

2.4.1.3.5. Educational Applications

- `cmututor` – an implementation of an interactive programming environment for a version of the **Tutor** computer-aided instruction language, as developed by the Plato project at the University of Illinois. It is one of the world's less elegant programming languages, but there are about 10000 hours of courseware for Plato..... The system uses documents to display text and edit the program; it is based on incremental compilation into P-codes for rapid execution and rapid editing.
- `graph` – a "playground" for experimenting with families of equations (including ODE's) and viewing the resulting graphs. The equations are specified in a document; the idea is for the student to read a text in one window, cutting the equations out and pasting into `graph` to experiment with them.
- `optics` – a simulation of an optics bench.
- `orbit` – orbital mechanics in a double-planet system.
- `gt` – a simulation of a simple mechanics experiment.

2.4.1.3.6. Miscellaneous

- `preview` – takes DVI `troff` output and displays it in a window at close to full size with every character correctly positioned. This saves many trees.
- `clock` – the ubiquitous window manager application. This one is both analog and digital.
- `gvmstat` – a graphical version of vmstat. An essential tool to answer the fundamental question of window systems – *why is it going so slowly?*
- `console` – the proliferation of tools like `clock`, `gvmstat`, and so on got out of hand. So Nathaniel Borenstein built `console`, a generalized window daemon that accepts a Scribe-like declarative specification of a layout of instruments such as graphs of performance variables, clocks, console logs, mail notifiers, and so on. It is capable of displaying consoles ranging from the Cadillac to the Boeing 747 (which takes 20% of a Sun/2).
- `fill` – fills the window with a pattern, useful as window-paper.
- `go` – a two-player go game, with windows on two machines.
- `talk` – a multi-player conversation, with windows on many machines.
- `zip` – a browser for pictures stored in a database. It is being used to present census data at a parish level for a history course.
- `zit` – a hyper-text editor, which is being used as the basis for a note-carding system in use for a creative writing course.

2.4.1.4. The Utilities

Andrew comes with a large set of utilities, mostly tools for mapping between various types of file (troff, Scribe, C, etc.) and the internal document format.

2.4.2. Facilities of the Andrew Server

The Andrew server provides two sets of facilities in one program. It supports an RPC interface by which clients may obtain and use windows, and a user interface by which users may manipulate windows.

2.4.2.1. The Client Interface

The normal object manipulated by the client is a window, representing some screen space (or perhaps none if it is hidden). Parts of a window can be saved and restored in the server, but the client can never see the bitmap representation of any part of a window.

- Open/Close Window - a heavyweight operation involving establishing a new connection to the server, and obtaining display space. Programs may have windows up to the limit on file descriptors, but almost all operate with a single window. The server provides almost no support for subdividing this window. Programs can hint about the size of windows, but must be prepared to deal with windows of any size (or none, which is what a hidden window looks like to the client).
- Window attribute operations - attributes include title bar information, clip regions, cursor regions.
- Output operations - line, trapezoid fill, RasterOp. These apply to windows and not to stored bitmaps. Output is always clipped to the window boundary, but a more restrictive clip may be specified.
- Save & Restore operations - allocate and free storage in the server for bitmaps, copy bitmaps between storage and parts of windows. Bitmaps can *not* be copied between client and server, Thus icons, for example, must be defined as characters in special icon fonts.
- Cursor operations - define cursors associated with parts of windows. Constrain cursor within parts of windows (used by scroll bar).
- Font operations - open and close fonts, write character strings with specified inter-character and inter-space shims in specified fonts. Andrew's font representation handles kerned, proportionally-spaced fonts with arbitrary baseline directions. Fonts are named by strings, such as `Helvetica10bi`, encoding their properties. When such a font is opened, the server scans the fonts it knows and selects the closest approximation to the specified set of properties. It then sends the client a specification of the bounding box and spacing parameters of each character. Some approximation will always be made; opening a font can never fail.
- Cut-buffer operations - put and get byte strings to a ring of cut-buffers.
- Event operations - Andrew supports keyboard, mouse button, window boundary crossing, mouse motion, and input focus change events. They can be selectively masked. Keyboard events can be re-bound.

- Re-paint requests. The Andrew server notifies its clients whenever they are required to re-paint their window. Since windows tile, these events are generated only when windows changes size, but this fact is unknown to the client. The tiling policy reduces the frequency of re-paint requests but the clients would operate just as well with another policy. In fact, the design of the user interface has evolved to minimize the number of re-paint requests because this improves the performance significantly.

The Andrew client interface has been very carefully designed to have a number of properties:

- The only operation that can fail is opening a window. Once a window is open, nothing can possibly go wrong. The server will do something sensible or ignore the request, but it will never object. Part of the reason why Andrew has many clients is the peace of mind this engenders in application programmers; there is never any need to write code to handle errors.
- Almost no operations return values, since returning a value means blocking to await a response from the server.
- The client can never see bitmaps, not even the bitmaps defining characters in a font. The information describing how characters are drawn is private to the server, and thus may be device dependent. In this way clients are portable by default.
- All requests for resources, such as window size or fonts, are hints. This has two effects:
  - Clients cannot depend on particular resource allocation policies, and will thus survive changes in them.
  - Clients cannot wire-in knowledge about particular resources, such as the spacings in Helvetica 10-point fonts. On the other hand, they can safely wire-in the names of fonts, since they are only hints. Nothing will go wrong if the server can't find a 10-point Helvetica, it will just use some other font.

Note that a client has no responsibility for observing clipping restrictions or damage lists, but must re-paint its image when required. The server as currently implemented has no memory other than the display's pixels for the contents of a window. Additional memory could be used to reduce the frequency of re-paint requests still further, but this would be invisible to the clients.

A client may simply draw a fixed-size image, trusting the server to clip it at the window boundaries. In some cases this is the right thing to do, but most applications are under strong social pressure to cope better with size changes. All the Base Editor objects reshape themselves properly (text re-wraps, button arrays re-lay themselves out, the H19 emulator chooses a font size to fill the window), and this background makes clients that fail to account properly for window size changes conspicuous.

2.4.2.2. The User Interface

Andrew's ''tiling'' user interface has evolved through a large number of variations with two main goals:

- The interface should be simple enough to explain on a single sheet of paper.
- The system should feel very fast and responsive.

The goal of simplicity has led to "column-mode", a style similar to Xerox's Cedar, in which the screen is divided into a series of vertical stacks of windows. A new instance of a type of window, for example an editor, takes some of the gray space at the bottom of the appropriate stack. When windows vanish, the windows below them in the stack are shuffled up to concentrate the gray space at the bottom. Windows can be either *open*, with both a headline bar and some contents visible, or *closed*, with just the headline bar visible. Users can:

- Click the *left* button in a headline to flip the window between closed and open.
- Use the *middle* button to drag any of the bars between windows, causing the windows alongside to move or re-size as required.
- Click the *right* button in a headline to move the window elsewhere. The window vanishes, the cursor changes to a target, and a subsequent click positions the window, shrinking the clicked-on window if required.

This interface has been very well received; it is simple to explain and use, it supports all the operations needed without using menus, all windows are always visible in some form, new windows appear without user intervention in a predictable place, and it is very fast. The speed is gained by minimizing the number of times a client has to be requested to re-paint; almost all window manipulations need no re-paints, they involve only copy operations by the server to move images on the screen without changing their size. Even opening a window or dragging a horizontal bar normally requires only one client to re-paint; if there is gray space at the bottom of the column the neighbour can move out of the way.

2.4.3. Kernel Support Required by the Andrew Server

The server requires no kernel support beyond the ability to map the pixels or device registers of the display into its address space. This is achieved variously using `mmap()` on special files, by mapping them into all user processes, and other hacks. Although the server can make use of special mouse and keyboard drivers (on the uVAX it uses the same drivers as X), it will work quite normally with input from `stdin` and a Mouse Systems optical mouse on a normal TTY port.

Re-paint requests are generated by the server, and transmitted to the clients using the out-of-band signals of TCP. They arrive as asynchronous signals. An alternative synchronous notification scheme based on synthetic events could be used.

2.4.4. Porting the Andrew System

The Andrew server and clients now run on at least 8 different displays and 3 different CPUs. This experience has allowed the parts of the system that are dependent on displays or CPUs to be identified with some precision.

The server's RasterOp is clearly dependent on both the CPU and the display. Andrew runs on displays with autonomous RasterOp processors, on displays with a shift-&-mask data path like the original Sun monochrome, and on displays which are just memory like the Sun-2 and the QVSS. None of the displays used so far has required any assembly language programming; most use Bruce Lucas's generalized C RasterOp. This uses many layers of macros to hide differences such as pixel order and access alignment restrictions, and is very easy to port across displays that can be accessed as memory. It isn't

optimal on any of them, but by using loop macros that can be adapted to exploit, for example, DBRA on the MC68010 it is efficient enough to make further improvements a low priority. Using this, one is not dependent on either the delivery or the understanding of proprietary RasterOp code for a new display, and the server has been ported to a new display in less than five hours.

Porting the clients has never involved more than simple re-compilation. The protocol that passes along the IPC channels explicitly accounts for byte-swapping problems, and pixel-order problems do not arise because bitmaps are never transferred.

2.5. X

The X window system was developed with strong DEC support at Project Athena, MIT's DEC & IBM-funded campus computing project. It is owned by MIT, and will be distributed on the 4.3BSD tape. DEC are using it as the product window manager for Ultrix on the micro-VAX (uVAX). It is probable that the other IBM Advanced Educational Program Universities (Brown, C-MU, etc.) will migrate to using X.

X is a network window server; client programs make remote procedure calls via stream connections to the server to perform graphics operations and receive input events. X as supplied uses normal 4.2BSD TCP/IP streams, but it has also been demonstrated running over the 4.2BSD DecNET support.

X is conceptually similar to the Andrew system developed at C-MU, in that they both use similar RPC mechanisms and rely on their clients to repaint the display when windows are damaged or change size. MIT has committed to providing a compatibility library that will permit Andrew client programs to be re-linked to use the X server. The major differences are:

- The user interface and window layout policies of Andrew are wired-in to the server; those of X are implemented by one of a number of privileged clients, each implementing a different user interface style.
- The X server efficiently supports a hierarchy of subwindows, even very large numbers of them. The Andrew server supports only a one-level division of the display, all sub-structures within windows are a client responsibility.
- Andrew provides effective support for high-quality fonts, and backs this up with a user-interface toolkit based on a WYSIWYG-style editor for multi-font multi-style documents (similar to but much more powerful that Sun's text panels). X provides only primitive font support and no editor or other higher-level support for text display.
- X fully supports color; Andrew does not.
- Andrew has been in production use since early 1984; X is just starting to get production use.

The following is a summary of the available information about the system, based on the Version 9 documents.

2.5.1. Components of the X System

2.5.1.1. The Server

The X server process supports the following configurations:

- VAX with Vs100 display (Unibus device with local MC68000 as graphics engine - no CPU access to bitmap).
- uVAX with QVSS display (display is just memory on the Q-bus).
- It is also believed to run on an un-announced DEC color display (called the QDSS?) on the uVAX, and to be being ported to an "experimental IBM workstation" at MIT and Brown.

The server implements graphic operations in a hierarchy of overlapping sub-windows. It tracks the mouse in user-level code with a cursor; the shape of the cursor is an attribute of a sub-window.

2.5.1.2. The Clients

The current X system provides only a few clients, all of which are known to run on VAX, uVAX, Sun 2, and "experimental IBM workstations":

- Bitmap (icon) editor.
- Analog/Digital clock
- Demo program
- Display a FAX (RFC 803) file in a window.
- Display all characters from a font.
- Impress previewer.
- Performance monitor (load average only).
- Window Manager - process providing user interface to window manipulation (e.g. hide, expose, resize). X supports several different styles.
- Terminal emulator - emulates a VT102 or a TEK4010, and provides cut-and-paste of characters.

2.5.1.3. The Utilities

The X system also provides some utilities:

- Keymap file compiler.
- resize - generates new size fields in termcap environment.
- Print a window on the DEC LN03 printer.
- Access control utility.
- Defaults setting program.
- Window dump and undump (bitmap image to/from file).

2.5.2. Facilities of the X Server

2.5.2.1. Operations on Displays

- Open/Close Display - returns the root window of the display. This is a heavy-weight operation involving establishing an IPC channel to the server.

2.5.2.2. Operations on Windows

- Create/Destroy a Child Window - Create is given a parent's resource name and returns the child's resource name. *Note that these are very cheap operations, and individual applications programs may efficiently manipulate hierarchies of hundreds of windows.*
- Map/Unmap windows - make them visible in their parent, clipped to its boundaries. If all their ancestors are mapped, this makes them visible on the screen. Calls in this group also move windows in X, Y, and depth.
- Window attribute operations - attributes include background and foreground colors, borders, clipping modes, desired sizes, icon windows, mouse state.
- Save & Restore operations - allocate and free storage in the server for bitmaps, copy bitmaps between storage and windows, copy bitmaps between client and server address spaces (this is slow for large bitmaps).

2.5.2.3. Graphic Operations

X's graphic operations apply to a window, whose pixels are in the server's address space but for which the client has a resource name.

- Output operations - line, brush, fill.
- RasterOp, including a general operation with source and destination in a window and a pattern tile in the client's address space, and the ability to operate between a window and a bitmap in the server's address space.
- Cursor operations - define cursors associated with windows.
- Color operations - X supports both read-only and read-write color maps.
- Font operations - open and close fonts, write character strings with specified inter-character and inter-space shims in specified fonts. X's current font specification can deal only with characters with vertical boxes that do not overlap (i.e. no kerning, etc.). When a font is opened, the server reads the specified font file, and returns a structure defining the widths of each character to the client. Unlike Andrew, this operation does no font approximation and can fail.

2.5.2.4. Other Operations

- Access control operations - X maintains a list of machines from which it will accept connections.
- Cut-buffer operations - put and get byte strings to a ring of cut-buffers.
- Event operations - X supports keyboard, mouse button, window boundary crossing, mouse motion, exposure, unmap (hide), and input focus change events. They can be selectively masked, grabbed (to prevent access by other windows), and re-mapped.

2.5.3. Kernel Support Required by the X Server

The X server on the VAX uses two types of kernel support:

- The pixels (of the QVSS), or the device registers (of the Vs100), are mapped read/write into *all* Unix user processes, even though this mapping is only required by the server. This is easy to do on the VAX, the addresses mapped are in system space and all that is needed is to change the system space page table entries. Unfortunately, this renders any VAX system running X vulnerable to being crashed with a machine check (e.g. odd Unibus address) by a malicious or careless user program. This problem is a symptom of the fact that `mmap()` doesn't work.
- A special keyboard/mouse driver is used to avoid `read()` calls and to maintain strict ordering of events. It uses memory shared between the driver and the server (see hack above) to implement a circular buffer of events. The driver writes events to this FIFO when either key or mouse events occur; the server can either select() on the device or simply examine the queue pointers to see if there are events to process. The driver also maintains the current mouse coordinates in shared memory.

The re-paint requests to the client are generated by the server and inserted into the event stream, needing no special kernel support. Unlike Andrew's, they are synchronous. *Note that the client has no responsibility to observe clip restrictions, but must respond to re-paint requests either by re-painting the affected regions, or by re-painting the entire image and trusting to the server to enforce the necessary clipping.*

2.5.4. Porting the X System

X specifies a device-dependent interface within the server. This is at a fairly high level, since it was designed to exploit the intelligent Vs100 hardware. It consists of:

- a set of RasterOps on PixMaps (multi-bit per pixel) and BitMaps (single-bit per pixel), each taking a list of clip rectangles.
- a set of font selection and character painting operations, again clipped against a list of rectangles.
- a set of rectangle save and restore routines, with associated space management operations. There is no assumption that the CPU has direct access to either the displayed or the stored pixels.
- A set of cursor operations.

The QVSS display is not intelligent, and RasterOps must be performed by the CPU. DEC regards the low-level RasterOp code as proprietary, but Athena has filled-in the gap between the low-level code and the device-dependent interface. This code will be shipped, and will presumably assist a porting effort, but the details are as yet unclear.

The kernel mouse/keyboard driver is VAX-specific, though work at Brown is reputed to have converted it to something like a line-discipline. It appears possible to fake the driver in a portable way by `read()` -ing from the keyboard and mouse devices and maintaining the event queue in the server's address space.

X specifies that bitmaps visible to the client are stored in 16-bit words left-to-right across the scanline, with the least significant bit to the left. This is convenient for the VAX and no-one else.

2.6. Other Window Servers

- The original UNIX window server in one sense was the Bell Labs Blit terminal. It runs a server-like system in a MC68000-based terminal, and has been described in several papers by Rob Pike. It works extremely well, even over relatively low-speed asynchronous links, because applications can customize the behaviour of the server by down-loading application specific code into the lightweight process in the terminal that is running their window.
- An alternative approach to a window server has been developed at Stanford. Their VGTS system is based on structured display lists, and offers higher-level abstractions as the basis for a virtual terminal than either X or Andrew.

2.7. Comparisons

This survey has covered two representative kernel-based window systems, and two window servers. A number of comparisons are worth drawing between them:

- SunWindows depends upon clients cooperating to a large extent (locking, clip lists, damage lists, re-paints). Andrew and X depend only on applications re-painting on request. Oriel hardly expects client cooperation at all.
- SunWindows needs locking for synchronization because the clients have access to the real resources. The others serialize accesses implicitly, Oriel because only the kernel can touch the real display, and Andrew and X because only the servers can.
- The Andrew server does not support sub-windows. SunWindows has a file for each sub-window, and is thus limited in the number of sub-windows a process may have. X and Oriel can manipulate very large numbers of sub-windows.
- SunWindows links the code implementing the user interface to window manipulations into each client. The others centralize this code in daemon or server processes. The location of this code can be critical for performance; the window manipulation code is used relatively infrequently but must feel very responsive:.
    - Andrew puts all window manipulation and menu code in the server. This process is highly interactive, being scheduled very frequently because it fields all the mouse and keyboard events as well as the RPC requests. It is thus unlikely to get paged or swapped out, and Andrew's window manipulation feels very fast.
    - X and Oriel place window manipulations in a daemon process that isn't fielding all the mouse, keyboard, and RPC requests. This process is highly interactive while it is in use, but has long pauses between bouts of activity. The UNIX scheduler provides quite good response for such processes, but in the inactive periods they may get paged or swapped.
    - SunWindows places window manipulations in each client. Apart from the swap space problems caused by this duplication, it virtually guarantees that the code needed to perform a particular window manipulation will be paged out when it is needed. Closing one window and opening another can involve paging the same code in twice!
- Porting SunWindows even between devices on the same CPU involves re-linking all the clients since they actually operate on the device. Porting Oriel between devices does not affect the clients, but porting between CPUs does, since they contain a

memory RasterOp that is CPU-dependent. Porting the other systems between displays or CPUs is easy; server-based systems do RasterOp only in the server.

- SunWindows and Andrew provide extensive user interface toolkits of pre-defined panel behaviours. Oriel provides a more limited set of such components; X provides none.
- As an example of (a) the code-sharing advantages of server-based systems and (b) the need for shared libraries in 4.?BSD, we can compare Andrew's `typescript` with SunWindows' `cmdtool`. Both implement a scrolling text panel interface to the shell or other programs, with cut-and-paste and other editing facilities:
  - `cmdtool` is about 400K of text, including drivers for all the Sun displays.
  - `typescript` is about 155K of text. Andrew's `wm` server is about the same size, including drivers for Sun 1 and 2 monochrome and Sun 1 color displays.

  Given shared libraries, both `typescript` and `cmdtool` would be tiny. They both involve only a few pages of code tying together pre-defined panels from the library.

3. Programmer's Model

The fundamental questions an application developer should be asking about window systems are *"what should I expect them to do for me?"* and *"what do I have to do in return?"* Reviewing the systems described above, we can start to answer these questions.

3.1. What You Get

In the early stages of the development of a window system, only the basic facilities will be available. Many systems get no further than this stage, and even at later stages the basic facilities are often the easiest to describe and use. The "XOR this rectangle with that one, then paint a 10-point Times Roman Italic H there" level of window system capability is important and fundamental, but it is *not* the level at which normal application programmers should be expected to work except in a dire emergency. The lower-level facilities of the system must be viewed as existing in order to make a user interface toolkit possible; all too often the toolkit is regarded as existing in order to disguise the chaos in the lower-level facilities. Viewed in this light:

- The system should provide a comprehensive user interface toolkit, so that applications can generally be constructed by pasting together pre-existing panels in suitable combinations.
- Since the library is unlikely to have all the panels required, the system must provide for the application to define its own. Doing so requires:
  - An efficient means for sub-dividing the window into panes.
  - A wide range of graphical output primitives for drawing in panes.
- The panels will need to process the input events directed at them. The toolkit should receive all inputs and distribute them to the panes, normally by calling appropriate routines in the panel implementation.
- The applications will need to share the real resources of the workstation, both the display and the input devices. The window system must provide protection and input demultiplexing.
- The user will need to be able to manipulate windows; moving, re-sizing, opening and closing them into icons, and pushing and popping them in the stack. Ideally, the system should be:
  - capable of laying-out new windows without user intervention
  - have user-replaceable layout policies
  - have user-replaceable window manipulation styles

3.1.1. User Interface Toolkit

A window system should provide a means of composing panels; being able to say things like:

> The top third of the window is an array of the following buttons: Abort, Quit, Exit, Terminate, Cancel, Shutdown. The middle third is an editable text panel with a scroll bar 25 pixels wide. The left half of the bottom third is a meter displaying keystrokes per second. The right half of the bottom third is a

read-only text panel with a scroll bar displaying the manual page.

The composition facilities should permit size specifications to be in terms of proportions of the window, absolute coordinates, and the size of a panel's contents.

It should also provide a wide range of pre-defined panels, including button arrays, sliders, meters, graphs, and above all text panels. In effect, in a system of this kind the text panel implementation replaces the old-style UNIX TTY driver as the primary mode of communication with the system. The text panel should offer:

- Speed – if it is to be an effective replacement for the TTY driver it cannot be too fast. Text output from programs running under a shell window should zip by much too fast to read. Typing in a text panel at full speed should not visibly affect the user or system time traces on the performance monitor.
- Fonts – the text panel should cope with multiple font families, in multiple sizes, with multiple attributes (bold, italic, underlined, and so on). It should support proportional spacing, kerning, and font changes at arbitrary positions in the text.
- Formatting – the text panel should support filling and various other justification modes, including left and right flush, and centering, between adjustable margins.
- Scrolling – the user should be able to page forward and backward, and to leap to an arbitrary position in the panel's contents. Ideally, the text should be capable of flowing smoothly in both directions at a fast scanning speed.
- Local editing – the text panel should support the conventional UNIX local editing functions, binding them to the keys specified in the *stty* command.
- Cut & Paste – the text panel should support selecting regions of the text, deleting them, and copying them to other text panels.
- Searches – the text panel should support forward and backward regular expression pattern searches, and some form of global replace operations.
- Undo – the editing operations on the panel should be undo-able.
- Printing – it should be possible to print out the contents of a text panel, both as they appear on the screen and re-formatted to fit the output page.

As regards user interface toolkits, SunWindows and Andrew score highly. There are some quibbles with both systems; SunWindows font support is inadequate and their text panels are too slow, and Andrew could do with an Undo capability and the ability to smooth-scroll backwards.

3.1.2. Drawing in Panes

To support the creation of application-specific panels, a window system should specify a panel interface to which they must conform, and should provide:

- An efficient means for sub-dividing the window into panes, and re-dividing those panes. Creating and destroying subdivisions should be very cheap; it should, for example, be possible to regard each item in a pop-up menu as a pane. In this respect X is outstanding, providing unmatched efficiency at manipulating large hierarchies of subwindows. The **Pixrect** level of SunWindows is also effective, though they are lighter-weight objects than X's subwindows.

- Comprehensive support for text output. All systems provide font selection and character and string output. But they vary in the detail of their font specifications, and in this area Andrew's support for multi-font proportionally spaced, kerned text with shimming and non-horizontal baselines is outstanding.
- A wide range of graphical output primitives for drawing in panes. All systems provide the basic rectangular RasterOps, pattern fill, and vectors. Additional primitives such as the trapezoid fill of Andrew are useful. In this area SunWindows stands out for its comprehensive set of output operations which include curve drawing and filling capabilities.

3.1.3. Input Facilities

The implementor of a panel must be able to nominate routines to handle the various types of input event, or to decide to ignore them. This means that the toolkit should receive all events directed at the window, and distribute them to the panels forming it, calling the nominated routines.

Transferring the input focus between the panels of a window is typically more complex than the simple area-based technique used to route events to windows; panels often wish to pre-empt others selectively on specific types of event (as for example with pop-up confirmation boxes).

To support these toolkit facilities, the window system should provide for all input events (keyboard up-down transitions, mouse motions, mouse button transitions, etc.) to be:

- serialized into a single stream in strict time sequence
- formatted into a uniform event report identifying the type of event
- stamped with the time of occurrence
- labeled with the mouse position at the time of occurrence
- distributed to the appropriate window

The strict serialization and the timestamping is required to support some user interface styles, such as double-click selection. Windows should be capable of ignoring events in certain classes, and perhaps of pre-empting all events everywhere. It should be possible for applications to generate synthetic events and insert them into the stream. In this area both Oriel and SunWindows provide adequate facilities, but X does not timestamp its events, and Andrew's facilities are even more primitive.

3.1.4. Protection and Sharing

The user will want multiple applications to share the real resources of the display and the input devices. The application should be unaware of this sharing, and to maintain this illusion the window system should:

- provide mutual protection between windows by enforcing appropriate clipping
- demultiplex the input devices, routing events to the appropriate window

All the systems provide adequate capabilities in this area. Even SunWindows, which depends on linking code into each client to perform clipping and locking, and lacks any real mechanism for enforcing protection, works well in practice.

For server-based systems, there is another aspect of sharing that is important. The server makes a workstation's screen into a resource accessible from anywhere in the network. Network accessibility was exploited in an early Andrew application, the *surprise server*. This was a daemon process that lived in the network†, and at intervals selected two machines at random. Using the facilities of `rsh`, it would cause the first of the machines to do something surprising on the screen of the second. A substantial surprise library was developed, including:

- The traditional `cookie` monster. A window would pop up and demand cookies, becoming more obstreporous as it was ignored. It could be pacified by feeding it cookies, either by typing their names or, more conveniently, by selecting different cookies from the pop-up menu.
- `tingle`, the random abuse program, modified to print each letter in a different font as if it had been cut from newspapers.
- The `logic bomb`, a window that ticked for a while in Russian, flashed violently, and then vanished.
- A program called `eliza`, that would (sometimes) respond conversationally when you typed in its window. The twist was that this program was actually the `talk` program in disguise; at the other end another person was also seeing an `eliza` window.....
- The `homeostat`, a program that selected the size of window it would like to have at random, and pestered the user with requests such as "make me taller" until it was satisfied.

Obviously, network access to windows is both useful and a source of problems. Some form of access control is required. Andrew simply ignores the problem, whereas X makes an attempt by allowing users to list machines from which they are willing to accept windows. Some more sophisticated access control mechanism is required, though it will probably have to be part of a more sophisticated control mechanism for remote execution in general.

3.1.5. User Interface to Window Manipulations

The user will need to be able to control the allocation of the real resources to the competing clients, by moving the input focus between windows, and by moving, re-sizing, pushing, popping, opening and closing windows. Again, the clients should be unaware of the mechanism for doing this, though they must respond to the results of the mechanism.

In most systems, there are actually two input foci. One controls the distribution of mouse (and normally menu) events, and the other the distribution of keystrokes. Typically the systems can be run in two modes:

- Focus Follows Cursor – in which both foci are tied together and follow the mouse.

† Or rather, two processes. One actually did the work and arranged to die at random intervals; the other monitored the first, and if it had died or been killed re-incarnated it elsewhere. At intervals, the second process would re-locate itself as well.....

- Click To Type – in which the mouse focus follows the mouse, and the keyboard focus remains where it was last placed until a mouse button is clicked in a new window.

The second style is normally used to support Xerox-style shift-select and its derivatives, in which text may be transferred to the text insertion point (even in another window) by selecting it with the mouse. This is the only mode available in Oriel, the other systems can operate in both modes.

The user should be able to replace the interface to window manipulations. In particular, it should be possible to implement both:

- overlapping and tiling window layout policies
- styles in which new windows are automatically assigned display space, and styles in which the user must explicitly assign the space.

Ideally, manipulation of the boundaries between panels in a window should be possible using similar mechanisms; it is often difficult to explain to users that they can drag this boundary but not that one.

In this area, both X and Oriel concentrate these user interfaces in daemon processes that can easily be replaced. Andrew implements a choice between several styles in code installed in the server, but if you don't like any of these styles life gets hard. Changing your mind about these aspects of SunWindows requires relinking all the clients.

3.2. What It Costs

In return for supplying these services to the client, the window system may expect some services in return. Typically these include responding to partial or complete redraw requests; if the client can be relied upon to repaint its image on request the server need not devote resources to preserving the image, and need not attempt to figure out what the correct image should be after windows have changed size.

Responding to re-paint requests is a significant responsibility. Many existing graphics applications are structured in such a way that there is no single routine that can be called to re-paint the image. Some applications may spend large amounts of time computing their images, and there may not be any simple data structure in which to keep the results of the computation to speed re-painting. In such cases, the window system should provide for a complete retained bitmap of the image, from which it can quickly be re-painted. SunWindows provides this option, and it is the only mode in which Oriel operates. The facility is lacking in Andrew, since although the contents of a window may be saved and later restored, this must be done by the client, and by the time the client finds out that the window has been damaged the damage has been done†. The facility seems also to be missing from X.

Clients should also respond carefully to window re-sizing. In almost all cases simply repositioning the origin at the origin of the new window and re-painting the original image through the new clip is the wrong thing to do. Systems such as Andrew, where text re-wraps itself, graphs re-scale themselves, titles re-center themselves, and so on

† A fix for this would be to associate a buffer with a window so that the server would save its contents before damaging it.

show that application-specific re-size behaviour is worth considerable effort.

4. Implementation

Given the requirements just set out, how do current window systems measure up? There are a number of problems:

- Completeness – no existing system implements the full set of facilities required.
- Performance – no existing system can be satisfied with its performance and responsiveness.
- Availability – the only system (Andrew) which has demonstrated that it is technically easy to port between workstations cannot be widely distributed because it is proprietary. Other systems have yet to demonstrate their portability, and most are also proprietary. Obviously, the advent of X is a hopeful development in this respect.

In this section, we examine a number of areas in which customer's demands and suggestions can help window system suppliers do the right thing. Of course, as far as X is concerned everyone can be a supplier, so this section can also be viewed as a list of things for which volunteers should sign-up.

4.1. Completeness

The major component that is missing from X, as is natural when a new window system is created, is the user interface toolkit. A generally usable toolkit would be a major step forward, as it would set a framework in which user interface components could be widely reused (see Tom Neuendorffer's paper at this conference *GLO - A Tool for Developing Window-Based Programs*). The design of such toolkits is now fairly well understood; all that is needed is for someone to bite the bullet of implementing it and giving it away.

Given the toolkit framework, easily the most important plug-in component is the text panel. Andrew is barely adequate in this area, and other systems are frankly deficient. Given the role of the text panel as a replacement for the TTY driver, and the extent to which UNIX has been plagued by poorly implemented and incompatible TTY drivers, it is hard to overstate the importance of an effective and widely accepted text panel. For some details on the specification of a suitable panel, refer back to section 3.1.1.

4.2. Performance

In the excitement about window systems it is easy to lose sight of the fact that they are only a tool for communicating with the real applications. No-one wants to run a window system for the sheer joy of dividing up the screen into overlapping rectangles. If the tool for communicating with the real applications takes 100% of the CPU, or if it is so slow that everyone ends up working through a single shell window using job control to multiplex tasks, the window system has failed.

There are a number of reasons for poor performance, some of which are easier to fix than others:

- Inadequate hardware.
- Window system designs that squander resources.
- Window system designs that interact poorly with the UNIX environment.

- Inadequacies in the UNIX environment itself.

4.2.1. Inadequate Hardware

I have pontificated on this subject before (at the 1985 Paris EUUG meeting); so this is only a summary:

- A workstation is only as good as its display, and the display is only as good as the bandwidth of communication between it and the CPU. In many cases, major computer manufacturers market as workstations hardware that was designed as a small computer with a memory-mapped I/O bus. This is then magically converted into a workstation by adding a display to the bus. Bandwidth to this bus is often an order of magnitude less than to system memory, with predictable results on display performance.

  Character drawing speed is a convenient measure of display performance. The overwhelming majority of RasterOps paint a character, so it is a good measure of overall system performance. Andrew provides an excellent test-bed, since in most cases the RasterOp code is effectively the same:

Andrew Character Drawing	
Workstation	Approx chars/sec
Sun/1	10000
Sun/2	6000
Sun/3	24000
uVAX II	4000

  The effects of the bit-addressable display memory on the Sun/1 (which doesn't use the same RasterOp as the /2 and /3) and the poor Q-bus bandwidth on the microVAX II (which does use the same RasterOp) are clearly visible.

- A cursor tracking the mouse is an essential part of a window system, and the lack of suitable hardware support for it can be very expensive in either performance of user interface quality. Lacking cursor hardware, the window system must paint the cursor into the real bitmap, and then interlock with the clients to ensure that it is removed before any RasterOp that overlaps it. Performing this interlock at a coarse granularity makes the cursor flash and track poorly, performing it at a fine granularity slows the entire system. A video-mixed cursor should be an essential part of a display design, as anyone who has tried the cursor tracking on the Whitechapel MG-1 would agree.
- Hardware designers are like hospital doctors, they *know* what is wrong with you and how to fix it. Obviously, RasterOp is an expensive but well-specified function, so it should be cast in silicon. Unfortunately, the attempts to do so somehow always cause more problems than they solve. Typically, they do too much in slightly the wrong way. An example is the way the Vs100 and similar displays require you to use the hardware designer's idea of a good font format. Another common failing is to forget that the RasterOp hardware must be used in a multi-process environment, and to make context-switching it expensive or even impossible.

4.2.2. Squandering Resources

When system designers first encounter workstations, they tend to believe that the days of resource constraints are gone forever. All that CPU horsepower and all that virtual memory! Mine, all Mine! The result is that they abandon all restraint, finding themselves doing things they would have been ashamed of a few short months before, and which they live to regret. Examples of this are:

- Waste of CPU – Andrew's initial window layout policy, which required every window to re-paint itself every time any window changed size.
- Waste of swap space – The way that the components in many window systems libraries are so interdependent that the library acts like a tar-ball; the slightest touch drags with it a vast weight of code that you never imagined you would need, and can't imagine what calls.

4.2.3. Poor Interaction with UNIX

There is often a clash between designer's ambitions for the user interface and the practicalities of the UNIX environment. The most frequent form in which this becomes evident is excessive context switches, a symptom of function having been divided among many processes, and this division having been forgotten. Examples are:

- the various attempts to do rubber-banding in the Andrew environment, where each mouse movement must be read by the server, transmitted to the client, the response transmitted to the server and then echoed to the display.
- the echoing of the transfer of the input focus by the individual clients in SunWindows.

Combining excessive context switches with the large processes often found in window environments in systems with limited real memory can rapidly confuse the pager and swapper, and lead to truly awful performance.

4.2.4. Inadequacies in UNIX

Jim Gettys' paper at this conference *Problems Implementing Window Systems in UNIX* explains in some detail the inadequacies as far as implementation is concerned. I have only to add some notes on the inadequacies as far as performance is concerned.

The major inadequacy in 4BSD systems that prevents window systems from performing as they should is that `mmap()` doesn't work. If it did:

- client programs could efficiently share the huge libraries of interactive techniques they depend on, reducing the demand for swap space and for page frames.
- RPCs between processes on the same machine (the normal case) could use shared writable memory instead of system calls.

The other inadequacy of 4BSD systems is the scheduler, swapper and pager. They have grown up to provide good interactive response on heavily-loaded timesharing systems, where there are typically many processes runnable at any one time. Blocking one while it takes a page fault or gets swapped back in is not a disaster.

In a workstation environment, there are typically very few runnable processes at a time. Furthermore, there is typically only one potentially interactive process at a time, the one with the input focus. A series of minor changes can improve the responsiveness of the

system considerably:

- The window system (if run as root) can raise the priority of the process group holding the input focus. The way that `csh` manipulates the process groups makes this a little tricky; typically the processes you really want are the process with the input focus (perhaps `typescript`), and processes in its process group (normally `csh`), and process groups whose leaders are children of these processes.
- Now that processes with raised priority levels are likely to be those with the input focus, the swapper can be modified to be more reluctant to swap out processes with raised priority, and more anxious to swap them back in.
- The pager can also be modified to make processes with raised priority less likely to suffer erosion of their in-core page sets. At present, the niceness of the process(es) holding the page have no effect on the decision to free it.

These changes can also help processes, such as the X and Oriel daemons, that are critical to system responsiveness but only get run infrequently.

4.3. Availability

There are a number of areas in which the availability of window systems could be improved, principally by making X or its successors easier to port and distribute:

- The key to Andrew's portability is Bruce Lucas' efficient C memory RasterOp, which isolates key machine dependencies in layers of macros. X would be much more portable if analogous code were available in the 4BSD domain. The Andrew code is only copyright, not trade secret, so there is nothing to stop someone from learning from Bruce's example.
- A window system, particularly one with pretensions to adequate text handling, requires a large collection of high-quality display fonts. Andrew includes a large collection of low-quality fonts; they are public-domain although the format in which they are stored is not. Effort devoted to developing a better font representation for X, tools for manipulating fonts in such a format, and either hand-tuning the TEX-derived fonts or producing an alternative public-domain set would pay handsome dividends in window system availability.

5. Writing Portable Programs

Unfortunately, the new facilities outlined in the previous section will take a while to arrive. In the meantime, the application developer is faced with a number of only partially satisfactory and incompatible window systems. How to make the best use of the one you've got, while being suitably defensive about the possibility that it may go away and be replaced with another?

5.1. Hints

There are a number of hints that will help you prepare for the day that someone sweeps your current window system into the dustbin of history and presents you with a much better and just slightly incompatible one:

- Don't even *think* that you have the actual screen pixels in your address space. Building your program around this assumption will be fatal if you ever have to move to a window server.
- Be prepared to re-paint your image on request. Better yet, be prepared to re-paint arbitrary rectangles in your image on request. There are very few systems that will never ask you to re-paint.

  If your application invests a lot of computation in its image, try to find some way of representing the image for rapid re-painting. Depend upon the window system being able to save a complete bitmap of the image only as a last resort.
- Regard all your requests for resources as *hints*. Always ask for some resource and then find out how much you actually have. Things in this category include window size, and the details of fonts. Never assume that just because you asked for an 12-point fixed-width bold font that the font you have obtained is 12-point, or because you asked for a space 80 times the character width across that you can put 80 characters into the space you were given. Be prepared to deal with ludicrously small or large amounts; if necessary emulate the homeostat and ask the user to give you more space with a polite "please make me bigger", or "please select a smaller font".
- Don't depend on user interface styles such as double-clicking or rubber-banding that require very close coupling between the mouse and the application. This isn't to say that they shouldn't be useable where they can be supported, but alternatives that don't need such close coupling should be provided too. And its always better to encapsulate these operations so that they can be easily replaced.
- Think hard before depending upon detailed properties of the filling and curve-drawing primitives your system provides. There is much less agreement about these than about the simpler primitives. For example, a major problem in providing a Mac-like interface is that Mac applications depend upon the precise way in which "circles" are not in fact circular.

5.2. Your Own Toolkit

If your window system provides a user interface toolkit, use it. Not merely because it will save you lots of time, but also because it is a good defensive tactic. Since libraries will typically contain similar panels, porting toolkit-based applications will usually consist to a large extent of choosing replacement panel types, and smoothing over the rough edges.

If your window system doesn't provide a toolkit, you can create your own quite easily. There are two components:

- The core, that composes panels, assigns screen space, and distributes input events to panels.
- The Panel interface, through which the core sees the panels. This defines the data available about each panel, and the operations that the panels may be called upon to perform.

A typical panel interface will be represented by some data structures, one defining an instance of a panel:

```
struct Panel {
    struct Point      origin;
    struct Point      size;
    int               depth;
    WindowHandle      resources;
    struct PanelOps   *operations;
};
```

The meanings of the fields are:

- the `origin` is typically the top left corner of the panel in whatever coordinates the output operations need
- the `size` is the width and height
- the `depth` a priority used for input distribution and perhaps for overlapping,
- the `resources` are whatever handle the window system needs to be given when drawing in this panel

the `operations` define the type of the panel by pointing to a structure like:

```
struct PanelOps {
    int               (*keystroke)(/* pan, chr */);
    int               (*mouseclik)(/* pan, but */
    int               (*mousemove)(/* pan, pos */);
    int               (*menuselct)(/* pan, mid, item */);
    int               (*updateimg)(/* pan */);
    int               (*repaint)(/* pan, rect */);
    Point             (*sizesugst)(/* pan, pl */);
    struct Panel      (*create)();
    int               (*destroy)();
};
```

where the operations are:

- `keystroke` if present, is called for every character typed into the panel
- `mouseclik` if present, is called for every button transition in the panel
- `mousemove` if present, is called for every mouse movement in the panel
- `menuselct` if present, is called for every menu selection in the panel
- `updateimg` if present, is called to incrementally update the image of the panel
- `repaint` must be present, and is called to repaint the image of the panel
- `sizesugst` if present is called with the size the core would like the panel to be, and the panel can reply with a different size (this is useful for fixed-size objects, and objects like button arrays that can re-size themselves in discrete increments. Of

course, the panel cannot *depend* on the core assigning the requested space)

- `create` is called to make a new instance of the panel
- Pg destroy is called to destroy a panel instance.

As an example, consider implementing a switch panel. The `create` routine would return a structure like:

```
struct SwitchPanel {
    struct Panel      panel;
    int                              state;
};
```

Note that its first field is a generic `Panel`, allowing pointers to `SwitchPanel`s to be given to the generic panel routines, and the rest of the structure is private state. The `mouseclik` routine would look like:

```
SwitchMouseClik(pan, but)
struct SwitchPanel *pan;
{
    pan->state = !pan->state;
}
```

Note that it knows that the `pan` argument will actually point to a `SwitchPanel` (you can include a `type` field in the `Panel` structure to make sure of this). The `repaint` routine would look like:

```
SwitchRePaint(pan)
struct SwitchPanel *pan;
{
    SetToBackGround(pan->origin, pan->size);
    if (pan->state)
        SwitchDrawOn(pan);
    else
        SwitchDrawOff(pan);
}
```

clearing the assigned space and drawing the switch in the appropriate state. In this case, the other fields are not needed, and can be `NULL`.