A Network Window-Manager

James A. Gosling
David S. H. Rosenthal

Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

CR Keywords: User interfaces, distributed/network graphics, graphics packages, window managers, remote procedure call.

An experimental window-manager for bitmapped displays has been developed that exploits 4.2 Berkeley Unix's interprocess communication mechanism, and the DARPA TCP/IP protocols. One window-manager process exists for each display; it tiles the screen with windows, and manages a mouse, keyboard and pop-up menus. Client processes make remote procedure calls requesting the window manager to create or destroy windows, and to draw text and graphics in them. The window manager asynchronously requests clients to re-draw their images when windows change size.

Introduction

You will get a better Gorilla effect if you use as big a piece of paper as possible.

Kunihiko Kasahara,
Creative Origami.

The charter of the Information Technology Center is to introduce a network of several thousand personal workstations to Carnegie-Mellon's campus. In preparation for this, an experimental window-manager for bitmapped displays has been developed that exploits 4.2 Berkeley Unix's interprocess communication mechanism@cite(4.2IPC), and the DARPA TCP/IP protocols. One window-manager process exists for each display; it tiles the screen with windows, and manages a mouse, keyboard and pop-up menus. Client processes make remote procedure calls (RPC) requesting the window manager to create or destroy windows, and to draw text and graphics in them. The window manager asynchronously requests clients to re-draw their images when windows change size.

A general font mechanism supports high-quality text output; bitmap, vector, and other font definitions may be used. Fonts are named by strings, for example *TimesRoman12*, and the window manager chooses the closest font available for its display. Clients may inquire the full range of parameters describing each font in each window.

The graphics interface uses pixel coordinates, and primitives such as RasterOp, line, and text. The RPC implementation chains and buffers these operations, so that most messages to the window manager are large. Using TCP/IP to transport the messages enables the window manager to operate gracefully in a distributed environment. For common operations, overall system performance is not much slower than one giving user processes direct access to the hardware.

All access to the display hardware is mediated by the window manager, insulating applications from its peculiarities@cite(CahnYen). This is essential; the campus network will be an open system, and applications must operate on incompatible workstations. The window

manager currently runs on a network of 20+ SUN workstations, using their 1000-by-800 pixel display, keyboard, and mouse. Porting it to future workstations will involve re-writing some low-level hardware interface routines in the window-manager process; applications will survive unchanged.

In the following a *user* is a person requesting services from a workstation. A *client* is a program requesting services from the window manager.

The User's View

The user of a workstation running the window manager sees a screen *tiled* with windows. The windows completely cover the screen, without overlapping one another. A user's initial window layout is specified in a profile file, though it may be changed at any time. Each window has a headline, containing an indentification of the client program attached to it, and the machine the client is running on. As the mouse is moved, a cursor tracks it on the screen, changing as it moves from window to window, or as clients use it as a feedback mechanism. Characters typed on the keyboard appear in the window containing the cursor, assuming the window is one to which it is sensible to type. Characters typed at the clock, for example, simply disappear.

When the mouse buttons are pressed various window-specific things happen; menus pop up, items are selected, objects move, and so on. Menus normally pop up on the down stroke of a mouse button. They appear near the cursor, overlapping the windows under them. The menu is removed, and a selection possibly made, on the up stroke of a mouse button. While the mouse button is down the menu item under the cursor is highlighted. Menus may be hierarchic: any menu item may have a submenu, which will pop up nearby when the cursor is in the parent. If the cursor is moved into the submenu, the process repeats.

Using a menu that normally appears when the middle button is pressed, the user can change the size of a window and reposition it. A window may be hidden; hidden windows appear as items in a submenu. New windows may appear on the screen at any time, as client processes request them. They cause exisiting windows to be re-sized or hidden at the whim of the window manager; creating a window does not involve interaction with the user. In fact, users cannot create windows directly; they can only create processes that will create windows.

The norm among window managers permits windows to overlap@Cite(Canvas, NUnix, BrownASH, SunWM, LisaWM, RobPike). Tiling window managers are rare@Cite(StarUI). Our decision to experiment with tiling was based on two observations of overlapping window managers in use:

- Experienced users typically lay out their screens so that the windows do not in fact overlap.

- Creating a new window is traumatic. The user has to point out opposing corners of the screen space to be allocated to the new window, and often adjusts several other windows.

It seems that only transient windows overlap others; the reason they do so is the disproportionate number of interactions needed to adjust the layout. In contrast, the constraint that windows must tile the screen allows the window manager, using heuristics, to allocate space for new windows and adjust the screen layout autonomously. The heuristics need not be complex, the user can override the window manager's decisions if the investment in interaction would pay off.

The Client Program's View

The client program interacts with the window manager by means of function calls, file descriptors, and signals. The design of this interface had three objectives.

- It should be as simple as possible. The role of the window manager is to arbitrate among competing requests for real resources, such as the screen, the colour map, and the mouse. Within the limits needed to protect others, clients should be free to use the resources they are allocated as they see fit. If clients want a complex interface, for example GKS@Cite(GKSReport), this can be built on top.

- Clients should regard their requests for resources as *hints*. The window manager will use its best efforts, but cannot guarantee to provide the requested amounts. This has two effects. The user, who after all has to look at the result, has the final word on resource allocation. Clients are more robust; at times when resources are short requests typically succeed, at least partially, instead of failing. The client can use the resources allocated to ask the user for more, for example by displaying a plea to "make me bigger". Clients prepared to deal with partially satisfied requests are also more likely to survive changes in the underlying hardware.

- Existing programs needing only simple terminal I/O (i.e. those without *ioctl()* calls) should need no changes to run under the window manager. Terminal-oriented programs should either be able to use an intermediary process emulating a specific terminal type, or make appropriate changes to use the window manager directly.

Clients see windows as rectangular spaces of integer coordinates in which they can draw. The window manager ensures that the origin of the space is at the top left of the allocated screen space, and that output outside the rectangle from the origin to (*Xsize,Ysize*) is clipped away. The client may inquire the values of *Xsize* and *Ysize* if it wishes to adjust its image to fit the allocated space.

The client draws using a functional interface. Examples of the operations available are:

w = DefineWindow(h)	Creates a window *w* on host *h*.
SelectWindow(w)	Makes *w* be the current window.
SetFunction(op)	Sets the combination function for subsequent operations.
DrawTo(x,y) *and* MoveTo(x,y)	Moves the current position to (*x,y*), with and without drawing a line on the way.
RasterOp(sy,sy,dx,dy,w,h)	Performs "*d* = *d* op *s*" for every corresponding pixel in the rectangles whose upper left corners are (*sx,sy*) and (*dx,dy*), both (*w,h*) wide and high. Op is an arbitrary boolean function specified with SetFunction. The source and destination rectangles may reside in bitmaps on the screen or inuser memory. See @Cite(NewmanSproull).
RasterSmash(dx,dy,w,h)	Is a special version of RasterOp for situations where only the destination rectangle is needed.
f = DefineFont(n)	Defines a font whose name is the string *n*.

SelectFont(f)	Selects the defined font *f* for subsequent text drawing.
c = DefineColor(n)	Defines a color whose name is the string *n*.
SelectColor(c)	Selects the defined color *c* for subsequent drawing. These functions support coloured primitives; another mechanism gives access to colour indices and the corresponding colour map entries to support coloured images.
GetDimensions(& x,& y)	Sets *x* and *y* to the dimensions in pixels of the current window.
SetDimensions(minX,maxX,minY,maxY)	Hints to the window manager about the desired size of the current window.

DefineWindow takes only one parameter, the host on which the window is to be created. It returns a handle giving access to the window's resources. If the window is on the same host, the client may choose to do graphics output via memory-mapped access to the device registers or via the communication channel to the window manager. The choice is made by loading a different (and bigger) library; it is not visible at the interface.

Many other parameters affect the initial allocation of resources to a newly created window. Conceptually, they are all defaulted, and the client must use the normal window manager calls to request changes. In practice, the window manager uses lazy evaluation of window creation, so that change requests made early in a window's history are likely to affect its initial allocation. Clients need not know the complete parameter set, an important consideration in an evolving system.

Clients are expected to deal with windows of any size. The size specification given in SetDimensions is only a hint. At any time, either the user or the window manager may decide to change the size of the window, disregarding the hint. Clients allocated windows they cannot readily use are free to ask the user to change their size.

A client may have many windows, but the window manager calls affect only the *selected* window. Newly created windows are selected automatically, but other windows may be selected at any time. This technique is common in graphics systems@cite(GKSReport,COREReport); it allows complex lookups without severe performance costs and shortens parameter lists (and thus messages).

DefineFont takes as its only parameter the name of the font to be defined. For example "TimesRoman12b" defines 12 point Times Roman bold. If the font library doesn't contain exactly the required font, something "close" will be substituted. For example, "TimesRoman12" may be substituted for "TimesRoman12b" if no boldface Times Roman exists. It returns a handle that can be used to *select* the font.

The value returned by DefineFont is actually a pointer to a structure that describes in great detail the properties of the font. It is important to note that fonts are window specific. "DefineFont("*TimesRoman12b*")" in two different windows might return two different values if, for example, the windows were on two displays with different properties. Clients must use information from the font structures in order to position strings Constants are likely to bring disaster.

Text is output using the normal *write* system calls on a window's file descriptor. It appears

at the current (x,y) position using the selected font. There are several different routines for drawing strings relative to a positioning parameter. These apply only to the selected window. For example:

```
DrawString(WindowWidth/2,                                    WindowHeight/2,
    BetweenLeftAndRightBetweenTopAndBaseline, "Center");
```

draws the string "Center" centered vertically and horizontally in the current window.

The window manager notifies the client whenever the size of its window has changed, and the client is expected to re-draw its contents. Notification could be either *synchronous*, via synthetic input events, or *asynchronous*, via a Unix signal. Only asynchronous notification is currently implemented. The window manager makes no attempt to preserve the contents of the window being resized. There are two reasons for this apparent lack of courtesy:

- Different clients need to respond to re-sizing in different ways. For example, an editor may reformat a document to suit the new space, a clock may center its face in the window and scale it to be as large as possible, or a chart may rescale its borders and show more detail.
- Window managers that save window images often develop clients that don't cope adequately with re-sizing. Insisting on re-drawing forces clients to deal with re-sizing too.

It can be claimed that insisting on re-drawing makes client programs more difficult to write, but experience with this and other systems@cite(RosenthalSeattle) does not support the claim. Programs need to be re-structured, moving the code that initially calculates scales and draws borders to a procedure that can be re-executed. It's more a matter of discipline than of extra work.

The Implementation

The client interface was designed with the idea of implementing it by mapping the bitmap and the display device registers into each client process. Unfortunately, the SUN 1.5 hardware we had could not save and restore the display registers on context switches, so the display could be mapped into at most one process. Thus, the window manager is currently implemented as a single user process that communicates with the screen, mouse, keyboard and all the clients. We use the SUN-supplied device driver to map the bitmap and the display device registers into the window manager's address space. All other I/O uses standard Berkeley 4.2 BSD system calls. No kernel changes were needed.

The window manager process maintains the state of all windows, performs all the primitive graphic operations, receives all mouse inputs, and routes keystrokes, menu selections, and re-draw requests to the clients. It communicates with the clients via a remote procedure call mechanism implemented using 4.2 BSD sockets.

@Begin(Figure)
@blankspace(3in)
@Caption(Window manager structure)
@End(Figure)

A *socket*, as defined in *4.2* BSD Unix, is one end of a communication path. It has an associated type, naming domain, and protocol. The type defines the I/O semantics of operations on the socket; we use *stream* sockets, which provide byte streams much like pipes. Every socket has a name and a protocol in some domain; we use the Internet naming domain and the TCP/IP protocol for compatibility with other machines. A socket may be connected to another socket having the same domain/protocol pair. A connection between sockets within one machine is much like a named pipe in other versions of Unix. In fact, 4.2 implements pipes as connected pairs of sockets. All window manager calls in the client turn into messages sent via these pipe-like connections. The socket mechanism is especially elegant in that it makes intermachine boundaries transparent. Neither the client nor the window manager really

know whether there is a machine boundary between them.

When the client requests the creation of a window a communication path is set up between the client and the window manager. The window manager creates a structure in its address space which describes the properties of the window. It doesn't actually create the window until a request is recieved that requires its existance. This allows the window creation heuristics to use any information sent to the window manager in the intervening time.

The window creation heuristics are primarily based on four parameters: the minimum height and width, and the "preferred" height and width. We have tried several sets of window creation heuristics. The most complex, and one of the shortest lived, involved considering each window to be a rectangular frame with springs holding the sides apart, then performing a relaxation of a system of equations to minimize the energy in the compression of the springs. This was very uncomfortable to use since it almost always completely rearranged all windows each time a new window was created. The present heuristics will pick one window and split it to give some of its area to the new window. The window to split and the position and orientation of the split is based on minimising an error function that attempts to balance areas and preserve aspect ratios.

Character drawing is the most frequent request to the window manager. The performance of this operation is crucial. Just as the remote procedure call mechanism batches together window manager requests, character drawing requests are batched together. The lowest level routines that draw characters then recieve long strings. This allows the precomputation costs to be spread over many characters: clipping is done based on strings, not characters; and some RasterOp setup is removed from the inner loop.

A *font* as used by **DefineFont** and the character drawing routines is broken into two parts: some general information about the font and an array of *icons*. The general information includes the name of the font and a maximal bounding box for all of the icons in the font. An *icon* is a drawing, in some representation, that may be placed on the screen. Usually these icons correspond to the shapes of characters from the ASCII character set, but they need not. There are many representations possible for an icon. For example, they may be described by a bitmap specially aligned for the SUN hardware or they may be described by a set of vectors. Icons are split into two parts: the *type–specific* part, and the *generic* part. The type–specific part contains all the information that depends on the type of representation used for the icon, while the generic part contains the information that is independant of the representation. When a client program defines a font, the information returned to it contains only the generic information for each icon in the font: all of the type–specific information is eliminated. This allows the window manager to implement a font in a way that is tuned to each type of device while insulating clients from the differences and still allowing them sophisticated access to the properties of the font.

@comment{**The shim concept, and why it is a performance win.}

There is a limit on the number of clients that the window manager may have that is imposed by the limit on the number of open file descriptors. Each socket accessible by a Unix process uses one file descriptor. Typically, Unix processes are limited to 20 file descriptors. The implementation of **4.2 BSD** Unix allows this limit to be increased by recompiling the kernel, but the limit cannot be extended beyond 31. Normally this limit isn't a problem since having more than a half dozen windows visible on the screen looks cluttered and confusing. Unfortunatly, hidden windows also take up file descriptors and having too many of them can cause problems. We don't know of a satisfactory solution to the problem. One possibility that we considered was to use connectionless datagram sockets which would have meant that the window manager would only need one socket on which to recieve from all of its clients. The problem with these is that at present only unreliable datagrams are implemented: datagrams

get through with some probability between 0 and 1, exclusive.

Assessment

Because the interface is at a very low level, the window manager provides clients with very few services. Fortunately, this has not deterred people from writing client programs. The requirement to respond to re-draw requests has not caused problems. For many clients, the use of clipped and shifted pixel coordinates is natural and efficient. We expect also to provide a higher-level interface, supporting floating-point coordinate spaces, by means of a library.

When we discovered that we would have to implement the window manager using interprocess communication (IPC) rather than direct access to the hardware, we expected the performance to be unacceptable. Traditionally, the use of IPC in Unix has had severe performance penalties. But, to our surprise, the performance of the window manager is respectable. Common operations, such as drawing text, scrolling and clearing windows are almost indistinguishable from the same operation performed directly on the hardware. Line drawing, and small RasterOps between the screen and process address space also work well. Drawing grids of individual points, and large RasterOps between the screen and a process run much too slowly.

Window manager calls which return values and therefore require a full handshake between the two processes typically take about 19ms each if both the client and the window manager are on the same machine. This only slows to 22ms if they are on different machines. Very few operations require a full handshake and the rest are generally much faster. For example, the Emacs text editor takes 360ms to completely redraw the screen when it is run on a SUN with direct access to the display. The same test under the window manager but displaying the same text with the same font and using the same amount of screen area takes 530ms. This is 47% slower, which is hardly perceptible. Performing this test again with Emacs and the window manager on different machines yields an interesting result: the full redraw takes 390ms. Only 8% slower —the two machines are being effectivly used to divide the computational load.

One of our applications is an editor, similar in spirit to Bravo@cite(Bravo) or LisaWrite@cite(LisaWrite), that continuously reformats the document being edited. It deals with kerned, proportionally spaced fonts, left and right justification and filling, reformatting the current paragraph at every keystroke. One might expect that this would be feasible only if the editor had full knowledge of, and an intimate association with the hardware. However, working via IPC channels and our window manager involves only a small performance degradation.

The reasons for the acceptable performance are simple:

- For the common operations, the number of pixels affected per byte transferred is large. This is true for character drawing.
- The underlying IPC mechanism –4.2 sockets and TCP/IP –performs very well.
- Remote procedure calls that don't return values get chained to following calls. The RPC mechanism builds large buffers of requests and avoids sending unnecessary messages.
- Few procedures in the window manager interface return values, and those that do are called infrequently. The common operations RasterOp, line drawing, character drawing and positioning do not return values. Typical interactions between the window manager and the client consist of a single message from the client containing many operations.

Eventually we would like to move to a hybrid implementation: one that uses direct device ac-

cess if possible, otherwise falling back on remote procedure calls. Even if direct device access is possible, most clients are unlikely to use it. Few need the extra performance, and loading the extra graphics library will make them much bigger. Above all, most will prefer to remain device-independent.

The use of remote windows has become very important to us. Among other uses, we have:

- A program similar to *write* that allows two users to communicate. It opens windows on each screen and passes messages back and forth between the two.
- A program for arbitrating a game of go. The players each have a window showing the board and game status summaries.
- A performance monitor that draws "EEG-like" traces of important system statistics. Most people run it on their own machine, but in our environment where workstations are diskless, the performance of the disk server is equally important. It is easy to run the performance monitor on the disk server, displaying its results in a window alongside the local performance monitor. Correlating the displays in this way has revealed several interesting performance-related problems.

Acknowledgements

Bob Sproull has been an invaluable source of advice. The other members of the ITC's User Interface group, Fred Hansen, Tom Peters, and James Peterson, rushed in where others feared to tread, and suffered the consequences.

Paper to be given at the Protext I conference
22-26 October, 1984
Gresham Hotel, Dublin
Usrat_sh mowno

An Editor-Based User Interface Toolkit

—

Abstract

A toolkit been constructed at the Information Technology Centre for building interfaces between users and programs. The host system is a high-power personal workstation running 4.2BSD Unix and having a large bitmap display. The toolkit defines a set of data types which programs can manipulate and a mechanism for automatically mapping these objects onto a display. Programs can manipulate these data objects and the toolkit takes care of updating the screen image appropriatly. Similarly, users can edit the objects and the program will be informed. The most powerful primitive data type is the *document*: building a text editor in the same class as Bravo or LisaWrite on top of it is almost trivial. The flexibility of this approach allows other tools, such as a help system, a mail system, or even just a command language typescript, to use these facilities.

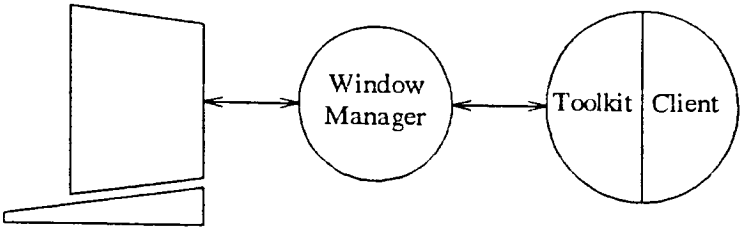
Introduction

The complexity of the implementation of computer user interfaces has been increasing dramatically as the power of computer systems has increased. The teletype view of the world didn't allow for much flexibility and hence forced simplifications. CRT terminals with their 24x80 matrix of characters led to simple graphical interaction techniques and such packages as *termcap* which captured many the common operations. High powered workstations with large fast bitmapped displays and graphical input devices (mice) are becoming very common. This gives programs the opportunity to have very sophisticated interfaces with dials, meters, buttons, menus, typeset documents, and many other types of objects. Such interfaces are typically very complicated since they have to support all of these types of objects and somehow weld them into a cohesive whole.

The toolkit described in this paper is an attempt to provide a general framework for building such interfaces and to provide an open-ended set of objects that can be placed into the framework. It has several goals:

- To lead to simple and uniform interfaces: all programs which use this toolkit should have interfaces whose various components act alike. For example, cutting and pasting should work similarly everywhere.
- To be simple to program: the system must be understandable to the programmers that are to use it.
- To be flexible: to allow new kinds of behaviours to be defined easily.
- To effectivly exploit the capabilities of the hardware.
- To perform well: There is a myth in computer science that performance is almost irrelevant when compared to functionality. In constructing user interfaces, a slow powerful system is almost always less useful than a fast simple one.

The system was built on a high power personal workstation running 4.2BSD Unix (SUN workstations). We built our own window manager through which the user interface toolkit performs all of its interactions. The components fit together roughly like this:



The window manager is a seperate process that mediates access to the display amongst the various client processes. Client processes may talk to the display directly with no assist, but usually the user interface toolkit is there to help. It sits between the client program and the window manager keeping screen images up-to-date, handling editing requests from the user and client, and informing each about the others changes.

The fundamental datatype is a *view*. A view corresponds to a rectangular patch of screen space in which a data object will be displayed. The data object may be a composition of other views or may be a primitive data object. Some primitives are defined in the toolkit and others may be defined by client programs.

Within a view is presented an image of a object. Objects may be integers, ranges (used for scroll bars), booleans, enumerations or documents. Other object types will eventually be supported as well. There are hooks in the system which allow one to define new types of objects and methods of representing them on the screen. The view manager takes care of all I/O, displatching mouse hits and characters, performing full redraws and incremental updates of the image.

An Example

For example, say that one wanted to write a card game whose interface contained a typescript of an ongoing computer-generated commentary, a few buttons having to do with actions in the game, and two hands of cards. The predefined document datatype could be used for the typescript, and the buttons would also use a predefined type. There isn't a predefined hand-of-cards datatype, so the client would have to define one. The definition of the interface could then look something like this:

```
typescript = DocumentView()
dealbutton = ButtonView(DealHitProcedure)
foldbutton = ButtonView(FoldHitProcedure)
programhand = PrimitiveView (HandHitProcedure, HandRedrawProcedure, HandUpdateProcedure, HandSizeProcedure, data-object)
userhand = PrimitiveView (HandHitProcedure, HandRedrawProcedure, HandUpdateProcedure, HandSizeProcedure, data-object)
interface = AboveOrBeside (Beside (userhand, Above (dealbutton, foldbutton), programhand), typescript)
```

DocumentView creates a *view* object and a *document*. If the program inserts text into the document the effect of that insertion will eventually be reflected automatically onto the screen. *ButtonView* creates a *view* and a *button* and arranges to call the associated procedure when the button is hit. Buttons are actually slightly more complicated, they correspond to boolean variables. The **ButtonHitProcedure** is activated as part of a general mechanism for informing the program of user editing operations.

Object Implementation

Programhand and *userhand* are defined as views on user specified data objects. The behaviour of these data objects (and ultimatly, all others) is defined by five procedures:

Hit the action to be performed when the user clicks a mouse button over the object.

Redraw — the action to be performed when the image of the object should be completely redrawn. The redraw procedure will be passed the data object to be drawn and the rectangle in which it is to be drawn.

Update — the action to be performed when a change has been made to the object and an incremental update is to be performed. Information about the extent of the incremental update is derived from the data object. In simple cases, this degenerates to clearing the region and redrawing the object.

Size — answers the question 'if you were to be placed inside a rectangle with this width and height, what size would you really like to be?'. For objects of constant size (like labels) the answer is fixed and doesn't depend on the size of the region the object is being squeezed into. For very flexible objects, like views on documents, the disired size usually matches the size of the target. Some objects, like arrays of buttons, have more complicated size behaviours since their actual size may depend on the number of rows and columns they decide to break themselves into. The size procedure is called by the algorithm that juggles view sizes and placement.

InputFocus — handles characters typed to this view. This procedure is *not* a part of a view, rather it is generally used by the hit procedure for the view to set the global input focus.

Screen Update

An important design goal was the decoupling of screen update from object update. When an object is updated, its screen image is not immediatly updated. Rather, an invocation of the objects update procedure is scheduled. It is the responsibility of the implementation of the object to maintain the information necessary to do incremental updates. For example, the 'boolean' datatype maintains two fields *value* and *DisplayedValue*. When the client program sets the boolean, the *value* field is changed and an update is scheduled. When the actual update happens, *value* and *DisplayedValue* are compared and the screen image will be updated appropriatly if they differ.

All screen updates will be performed just before the program next blocks for input from the keyboard. Updates from a sequence of operations are thus batched and done together. This was done partly to make the design cleaner and partly to avoid the phenomenon one sees in some systems where complicated operations that depend on smaller operations result in many screen updates.

Screen Space Allocation

One of the most complicated algorithms in this system is the one that lays views out on the screen. A complicating factor is that the system runs under a window manager which allows the user to easily change the shape of windows. Tools must therefore be able to dynamically adapt to varying window sizes and shapes. The task of the layout algorithm is to take a set of views on primitive objects which has been composed into a hierarchy and allocate space to each, such that each gets enough space. Primitive objects are allowed to have variable sizes.

As an example of an object type which complicates the algorithm consider a set-of-strings where the object should be arranged as a grid of rows and columns. The width of a column and the height of a row is fixed, but the number of rows and columns can vary, so long as their product exceeds the number of strings. Acceptable shapes range from tall and skinny to short and wide, where the range of shapes is not continuous. It was considered undesirable to have the layout algorithm incorporate detailed knowledge about all different size behaviours.

The algorithm that is currently being used is a brute force search through an enumeration of the entire set of possible layouts. While this sounds as though it could take a lot of time, the

application of clever heuristics makes it possible to dramatically prune the search tree.

Documents

The goals for document objects were slightly unusual in this project. On the one hand, we wanted to be able to do fairly sophisticated typesetting with them. On the other, we wanted to be able to use them as components of user interfaces. The phrase *What you see is what you get* is often used to describe text editors that emulate the quality of typesetters on the screen, providing an exact one-to-one correspondance between the screen images and the printed page. Often this leads to screen images which are large and poorly spaced since they are emulating the size of a piece of paper and the higher resolution of a printer. We placed a heavy emphasis on the readability of documents on the screen and dispensed with an exact correspondance between the document on the screen and on the printed page. For example, when one asks for help a manual entry will be presented in a window correctly formatted for the size of the window. If one asks to have the entry printed, then it will be reformatted and printed using the line and page lengths appropriate for the printer. When a document is sent to a printer, the user has the option of seeing a preview of the printed document, laid out exactly as it will be on the printer.

Mutter on, McDuff...

The ITC Window Manager
Programmers Manual

James Gosling

This document is a guide for programmers to the facilities of the ITC window manager. It describes all that one needs to know to be able to write programs that produce graphical images in windows.

A *window* used by a client of the graphics package is a rectangular patch of pixels. The upper left pixel is at coordinate (0,0) and coordinates increase down and to the right. The width and height of the window may be interrogated by calling wm_GetDimensions. Performing operations at coordinates outside of the window area causes the operation to be properly clipped. One unit in the horizontal or vertical direction corresponds to one pixel. There is a *current* position which moves around as operations are performed.

This extremely simple coordinate system was explicitly chosen in preference to a more general one with full homogenous transformations. Many client programs are insensitive to which coordinate system is chosen: any one is good as any other; others have such specialized requirements that they need to do their own coordinate transformations themselves, anyway.

When the dimensions or real position of a window are changed by the user the window manager will request the client program to redraw. This appears to the client as an asyncronous call on a procedure called *FlagRedraw*. This procedure should only set a flag indicating that a redraw needs to be done, it should not actually do the redraw. This is because it is called asynchronously via the signal mechanism. Any system calls that were in progress at the time of the flagging will be aborted by the system and will return an error with errno==EINTR.

The window manager provides no other support for handling window resizing other than the notification described. This is because only the client program can know the *right* way for its image to respond.

To compile a C program, the invocation of cc that compiles a client of the window manager should have the switch "-I/usr/local/include" and the program should contain the line "#include <usergraphics.h>". When linking the program, include "-litc" in the list of libraries.

struct wm_window *wm_NewWindow (*host*)

Creates a window on the desired host. If *host* is zero and the environment variable 'WMHOST' is set then the window will be created on host WMHOST. Otherwise the window will be created on the current host. A pointer to the window is returned and that window is selected as the current one. If a window cannot be created, a null pointer will be returned. The window will be automatically be assigned space on the screen using magic: the user will not be queried.

wm_SelectWindow (struct wm_window *w)

Makes *w* be the current user window. All subsequent operations are performed on the current window. Normally, in a client program that creates

only one window no window selection need be done since *wm_NewWindow* does it automatically.

wm_MoveTo (x, y)
Move the current position to the point (x,y).

wm_DrawTo (x, y)
Draws a line from the current position to the point (x,y) and leaves the current position there.

wm_SetFunction (f)
Sets the rasterop combination function to *f*. This function is used whenever a graphics operation is performed. For example,

```
wm_SetFunction(f_black);
wm_MoveTo(0, 0);
wm_DrawTo(100, 100);
```

will draw a black line. The following permit functions to be expressed as Boolean combinations of the three primitive functions 'source', 'mask', and 'dest'.

f_black
f_white
f_invert
f_copy Only makes sense with wm_RasterOp (and it's probably the only opcode that does make sense with wm_RasterOp)
f_BlackOnWhite
f_WhiteOnBlack

wm_SetTitle (char *s)
Sets the title line for the current window to *s*. Each window has a title line at its top which describes the entity being viewed through the window. For example, if the client program is an editor, then it should be the name of the file being edited.

wm_SetProgramName (char *s)
Sets the program name field in the current window's title line to *s*. Normally, programs don't set this, instead they use the **program** macro to declare the name of the program. **Program(name)** is placed in the main program, somewhere where global variables may be declared. It sets up a data structure which gets used by **wm_NewWindow** when a window is created and by **getprofile** when a preference option is being looked up.

wm_GetDimensions (int *width, int *height)
Sets *width and *height to the width and height of the current window.

wm_SetDimensions (MinWidth, MaxWidth, MinHeight, MaxHeight)
Sets the *preferred* minimum and maximum height and width for the current window. This does not change the actual dimensions of the window, it only provides hints to the window manager to guide its automatic selection of window sizes.

wm_SetRawInput ()

Sets the current window into raw input mode: each time that the user types a character it will immediatly be shipped down to the client, rather than saving up a line and waiting for newline to be typed.

m_DisableInput ()
Disables input from the current window. If the user types in this window, it will be ignored.

wm_EnableInput ()
Enables input from the current window. Characters that the user types will be passed to the client program. This is the default state.

wm_RasterOp (sx,sy,dx,dy,w,h)
Performs a RasterOp operation using the current function. (sx,sy) is the origin of the source rectangle and (dx,dy) is the origin of the destination rectangle. (w,h) is the width and height of both rectangles.

wm_RasterSmash (dx,dy,w,h)
Performs a RasterOp without a source rectangle. The meaningful operations are f_black, f_white and v_invert.

wm_FillTrapezoid (x1, y1, w1, x2, y2, w2, f, c)
Fills the trapezoid with *x1,y1* as its upper left corner, *w1* as the width of the top, *x2,y2* as its lower left corner, and *w2* as its width. The trapezoid will be filled with character *c* from font *f*. If *f* is -1 then the default shape font (shape10) will be used. **Bogosity:** *f* is a font index, not a font pointer.

wm_ClearWindow ()
Clears (to white) the current window.

wm_SawMouse (int *action, int *x, int *y)
If you've see a *wm_MouseInputToken* on the window input stream then wm_SawMouse will return in action, x and y the event that occurred on the mouse and its coordinates at the time. *wm_MouseInputToken* is a simple character. When a client program is reading characters from its window input it should be checking for occurences of *wm_MouseInputToken* . If one is seen, then *wm_SawMouse* should be called to decode the action and coordinate information. The *action* parameter tells the client what sort of event has occurred. For an explanation of the interpretation of *action* , read the section on *wm_SetMouseInterest* .

wm_SetMouseInterest (mask)
Defines a mask which describes the mouse events in which the client is interested. The mask is constructed by or ing together several values from *MouseMask* .

MouseMask (event)
Constructs a constant which represents the *event* occuring on the *button.*

Values for	
button	event
LeftButton	UpMovement
MiddleButton	DownTransition
RightButton	UpTransition
	DownMovement

wm_SetMouseGrid (n)
Causes the window manager to report mouse movements only when the mouse moves at least *n* pixels from its last reported position.

wm_SetMousePrefix(char *string)
Causes the window manager to prefix all mouse position reports with *string* instead of **wm_MouseInputToken.**

wm_AddMenu (char *string)
Adds the given string to the menu for the current window. Each window has associated with it one hierarchic menu: menu entries may have submenus, which in turn may have yet more submenus. *String* consists of a series of comma separated entry names, followed by a colon, followed by the response string. The response string is the string which will be transmitted back to the client program when that menu entry is selected. For example:

```
wm_AddMenu('Directory,List:ls -lvn');
wm_AddMenu('Directory,Name:pwdvn');
```

The first call to *wm_AddMenu* will add an entry in the root menu named "Directory". This entry will have a submenu, and in that submenu will be defined an entry named "List". This entry will be a leaf of the menu hierarchy and will have associated with it the string "ls -ln". If the user selects "List" in the "Directory" submenu then this string ("ls -ln") will be transmitted back to the client program through its window input channel (winin).

The second call to *wm_AddMenu* simply adds a "Name" entry to the "Directory" menu and associates the string "pwdn" with it.

The colon and rresponse string may be omitted, in which case the menu entry will be removed.

wm_SetMenuPrefix (char *string)
Causes all following menu selections to be prefixed by *string.* The string defaults to the empty string.

wm_DisableNewlines ()
Normally when a newline character is written to a window it will cause scrolling if occurs at the bottom of the window. *wm_DisableNewlines* disables this behaviour — newlines will simply move the text pointer off the bottom of the window and subsequent text will be clipped and not displayed.

struct wm_window *CurrentUserWindow
A pointer to the currently selected window.

FILE *winout
An output file corresponding to the current window. Writing text here causes the text to appear in the window just as though it were a glass tele-

type. The text will be drawn with the upper left hand corner of the first character being placed at the current position. The current position will be left just past the upper right hand corner of the last character.

FILE *winin

An input file corresponding to the current window. Reading from here returns characters typed by the user while pointing at this window.

Text

struct font *wm_DefineFont (fontname) Defines a font, given a *fontname*, and returns a pointer to be used by subsequent **wm_SelectFont's**. *Fontname* is a string which names a font. For example, "TimesRoman10i" specifies Times Roman 10 point italic; "CMR10b" specifies Computer Modern Roman 10 point boldface. If the font specified doesn't exist, one which is "close" will be used instead.

wm_SelectFont (struct font *fontpointer) Causes the font specified by the *fontpointer* to be used for subsequent character printing. *Fontpointer* is created by calling **wm_DefineFont.**

```
wm_SelectFont(wm_DefineFont("TimesRoman10i"))
```

causes all further printing to use Times Roman 10 point italic. The reason for separating the **wm_DefineFont** and **wm_SelectFont** is to avoid having to do many repeated font lookups —the client program is expected to save the value returned from **wm_DefineFont** and reuse it.

wm_StringWidth (string, int *x, int *y) Finds the width of the given string in the current font in the current window. The x-width is returned in x, and the y-width in y. For normal left-to-right fonts, y will be zero and x will be the width of the string. The width is measured starting at the origin of the leftmost character up to the origin of the character which would immediatly follow the rightmost character.

wm_DrawString (x, y, flags, string) Draws the given string relative to the given coordinates, according to the flags. The flags control alignment of the string relative to the height and width of the string.

Height alignment options	
wm_AtTop	wm_AtRight
wm_AtBottom	wm_AtLeft
wm_AtBaseline	wm_BetweenLeftAndRight
wm_BetweenTopAndBottom	wm_BetweenTopAndBaseline

The flags argument is constructed by oring together one height alignment option and one width alignment option. Either may be omitted and wm_AtBaseline and wm_AtLeft will be taken as defaults.

A slight confusion is possible in understanding the differences between wm_BetweenTopAndBottom and wm_BetweenTopAndBaseline. The former properly centers, taking into account descenders (Like the tail on a lowercase 'y'), where the latter ignores descenders. The latter is

likely to be more aesthetically pleasing.

For example:

```
wm_DrawString (WindowWidth/2,
        WindowHeight/2,
        wm_BetweenTopAndBaseline|wm_BetweenLeftAndRight,
        "Don't Panic");
```

will print the string "Don't Panic" correctly centered in the current window using the current font.

wm_printf (x, y, flags, format, args) This procedure is similar to wm_DrawString except that instead of taking a single string as an argument, it takes a full "printf" format and arguments.

wm_SetCursor (struct font *f; char c) Sets the cursor that is used in following the mouse to character 'c' from font 'f'. The cursor will only have this shape when it is inside the current window.

wm_SetStandardCursor (c) Sets the cursor from the standard icon font (icon12). The following cursor names and shapes are defined in usergraphics.h:

```
wm_GunsightCursor
wm_CrossCursor
wm_HourglassCursor
wm_RightFingerCursor
wm_HorizontalBarsCursor
wm_LowerRightCornerCursor
wm_PaintbrushCursor
wm_UpperLeftCornerCursor
wm_VerticalBarsCursor
wm_DangerousBendCursor
wm_CaretCursor
```

You can look at the available standard cursors by typing:
samplefont icon12

wm_SetSpaceShim (n) Sets the space shim value to *n*. A *shim* is some padding that is added on to the right of a character. The space shim applies only to space characters. After calling **wm_SetSpaceShim** all following space characters will have *n* added to their width.

wm_SetCharShim (n) is similar to **wm_SetSpaceShim** except that it applies to all characters, including spaces.

Color

This section needs to be written. But first, we should figure out what we're doing...

Window Manipulation

wm_HideMe() Hide the current window. Has no effect if the current window is currently hidden.

wm_ExposeMe () Expose the current window. Has no effect if the current window is currently exposed.

wm_DeleteWindow () Deletes the current window from the screen. Further operations on that window will fail.

wm_AcquireInputFocus () Acquires the input focus for the current window. When the input focus is 'acquired' by a window, all subsequent characters typed by the user are send to that window. This will continue until the input focus is acquired by some other window. The acquisition of the input focus only affects characters typed by the user, it does not affect mouse hits or menu selections.

wm_GiveUpInputFocus () Gives the input focus back to the last window that had the input focus.

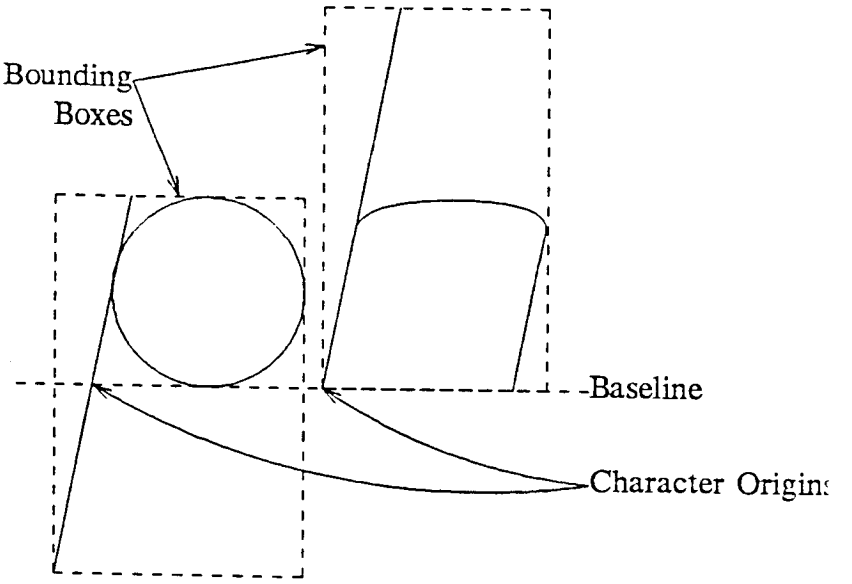
wm_IHandleAcquisition () Declares to the window manager that the client is willing and able to hadle input focus acquisition for himself. Normally, if no client is handling input focus acquisition, if the user wants to change the input focus he points the mouse at it and clicks a button. This mouse event will be thrown away and the input focus will shift. The user can then type at that window and use the mouse. If the client has executed **wm_IHandleAcquisition**, then all mouse clicks get sent to the client and the window manager will not automatically shift the input focus. That client can then (for example) acquire the input focus whenever it sees that first mouse event. The posession of the input focus will be encoded in recieved mouse events.

Fonts

This section describes the font mechanism used by the window manager. Most writers of client programs don't need to know much of what is described here.

A *font* is a collection of *icons* . An icon is a thing which can be drawn to make a mark. The mark can have any shape. A font has two parts: a header which contains the name of the font and some summary information, and an array of icons. An icon also has two parts: One of generic information and one of specific information. The generic information describes the properties of the icon that are independant of its represenation: information about the bounding box and spacing. The specific information describes the properties that are dependant on the representation of the icon —those that are interesting only to a routine that is actually drawing the icon. At the moment, the specific information will either be a bitmap or a list of vectors.

An icon represents some shape enclosed within a bounding box. Typically the icons are characters like those in this illustration. The bounding box for each character is at least large enough to completely enclose the character, and it is usually a tight fit. Within that bounding box is a distinguished point called the Origin. The origin is a point which is on the character's baseline and which is at the optical left edge of the character. The optical left edge of a character is often at the left edge of its bounding box. For italic characters with descenders, like the 'p' here it is inset from the left edge. The bounding boxes of successivly drawn characters may or may not overlap. The bounding box of an italic 'y' or 'p' will typically overlap the bounding box of the preceeding character so that their decender can sweep under it. Similarly, the ascender of an italic 'f' usually extends over the following character.



Associated with each character are five vectors:

NWtoOrigin is a vector which goes from the north west corner of the bounding box to the origin.

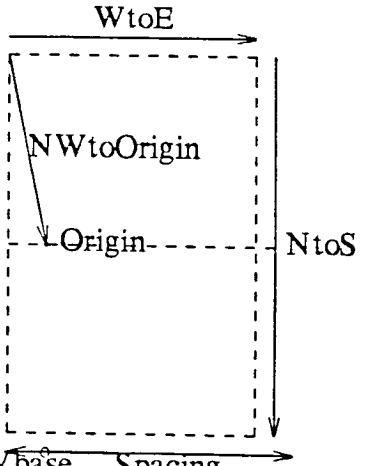
WtoE goes from the west edge to the east edge.

NtoS goes from the north edge to the south edge.

Wbase goes from the origin due west to the west edge of the character.

Spacing goes from the origin of the character to the place where the origin of the next character should be placed when this character is drawn with one following it.

These five paramaters could have been scalars, but were done as vectors instead in order to make manipulating rotated fonts easier.



A font file is laid out according to the following declarations. The file will start with a single instance of **struct font** . This contains the vector of icons. Each icon contains two offset pointers: one to the generic part and one to the specific part for the icon. All of the generic parts occur in the file before the specific parts. Charracters with matching generic or specific parts will share the corresponding space.

```
struct SVector {                    /* Short Vector */
        short x, y;
};

/* Given a pointer to an icon, GenericPart returns a pointer to its IconGenericPart
                                                                    */
```

```
#define GenericPart(icon) ((struct IconGenericPart *) (((int) (icon)) + (icon) ->
                        OffsetToGeneric))

/* Given a character and a font, GenericPartOfChar returns the corresponding
                        IconGenericPart */
#define GenericPartOfChar(f,c) GenericPart(& ((f)-> chars[c]))

struct IconGenericPart { /* information relating to this icon that is of general in-
                        terest */
        struct SVector Spacing; /* The vector which when added to the origin of
                                this character will give the origin of the next char-
                                acter to follow it */
        struct SVector NWtoOrigin;      /* Vector from the origin to the North
                                West corner of the characters bounding box */
        struct SVector NtoS;    /* North to south vector */
        struct SVector WtoE;    /* West to East vector */
        struct SVector Wbase;   /* Vector from the origin to the West edge paral-
                                lel to the baseline */
};
struct BitmapIconSpecificPart { /* information relating to an icon that is necessary
                        only if you intend to draw it */
        char type;                      /* The type of representation used for this
                                icon. (= BitmapIcon) */
        unsigned char rows,     /* rows and columns in this bitmap */
        unsigned char cols;
        char orow,                      /* row and column of the origin */
        char ocol;                      /* Note that these are signed */
        unsigned short bits[1]; /* The bitmap associated with this icon */
};

struct icon {                           /* An icon is made up of a generic and a
                        specific part. The address of each is "Offset" bytes from
                        the "icon" structure */
        short OffsetToGeneric;
        short OffsetToSpecific;
};


/* A font name description block. These are used in font definitions and in font
                        directories */
struct FontName {
        char FamilyName[16]; /* eg. "TimesRoman" */
        short rotation;                 /* The rotation of this font (degrees;
                                + ve=> clockwise) */
        char height;                    /* font height in points */
        char FaceCode;                  /* eg. "Italic" or "Bold" or "Bold Italic"
                                */
};

/* Possible icon types: */
#define AssortedIcon 0  /* Not used in icons, only in fonts: the icons have an as-
                        sortment of types */
#define BitmapIcon 1    /* The icon is represented by a bitmap */
#define VectorIcon 2    /* The icon is represented as vectors */
```

```
/* A font.  This structure is at the front of every font file.  The icon generic and
                    specific parts follow.  They are pointed to by offsets in the
                    icon structures */
struct font {
        short   magic;                  /* used to detect invalid font files */
        short   NonSpecificLength;      /* number of bytes in the font and generic
                                parts */
        struct FontName fn;     /* The name of this font */
        struct SVector NWtoOrigin;      /* These are "maximal" versions of the
                                variables by the same names in each constituent
                                icon */
        struct SVector NtoS;    /* North to South */
        struct SVector WtoE;    /* West to East */
        struct SVector Wbase;   /* From the origin along the baseline to the West
                                edge */
        struct SVector newline; /* The appropriate "newline" vector, its just NtoS
                                with an appropriate fudge factor added */
        char type;                      /* The type of representation used for the
                                icons within this font.  If all icons within the font
                                share the same type, then type is that type, other-
                                wise it is "AssortedIcon" */
        short NIcons;                   /* The number of icons actually in this
                                font. The constant "CharsPerFont" doesn't actually
                                restrict the size of the following array; it's just
                                used to specify the local common case */
        struct icon     chars[CharsPerFont];
         /* at the end of the font structure come the bits for each character */
};

/* The value of font-> magic is set to FONTMAGIC.  This is used to check that a
                    file does indeed contain a font */
# define FONTMAGIC 0x1fd

/* FaceCode flags */
# define BoldFace 1
# define ItalicFace 2
# define ShadowFace 4
# define FixedWidthFace 010

struct font *getpfont();  /* get font given name to parse */
struct font *getfont();   /* get font given parsed name */
struct icon *geticon();   /* get an icon given a font and a slot */
int LastX, LastY;               /* coordinates following the end of the last string
                        drawn */
```

Program Template

The following is a template for simple programs that use the window manager:

```
# include <usergraphics.h>
```

```
program(foo)

main () {
        wm_NewWindow ();
        ....
}
```

Sample Client Program

This is a simple clock which was written to test out user level graphics. It draws a face which consists of 12 tick marks arranged in a circle and the hour, minute and second hands. The clock face is updated every second.

```
#include <stdio.h>
#include "usergraphics.h"This library contains all of the definitions for the client in-
                        terface to the window manager
#include "clocktable.h"          This library contains a table of sines and cosines
#include <time.h>

struct tm LastTimeDisplayed;

int MidpointX,
int MidpointY;
int FaceRadius;
int RedrawRequested = 0;

FlagRedraw () {
        RedrawRequested++;
}

program (clock)

main () {
        int                     FaceInnerRingRadius;
        int                     n;
        int                     HourRadius, HourInner,
                                MinRadius,  MinInner,
                                SecRadius,  SecInner;
        register struct tm *CurrentTime;
        if (fork ())
                exit (0);
        FlagRedraw ();
        while (1) {
                long now = time (0);
                CurrentTime = (struct tm   *) localtime (& now);
                if (RedrawRequested) {
                        while (1) {
                                wm_GetDim& MidpointX, & MidpointY);
```

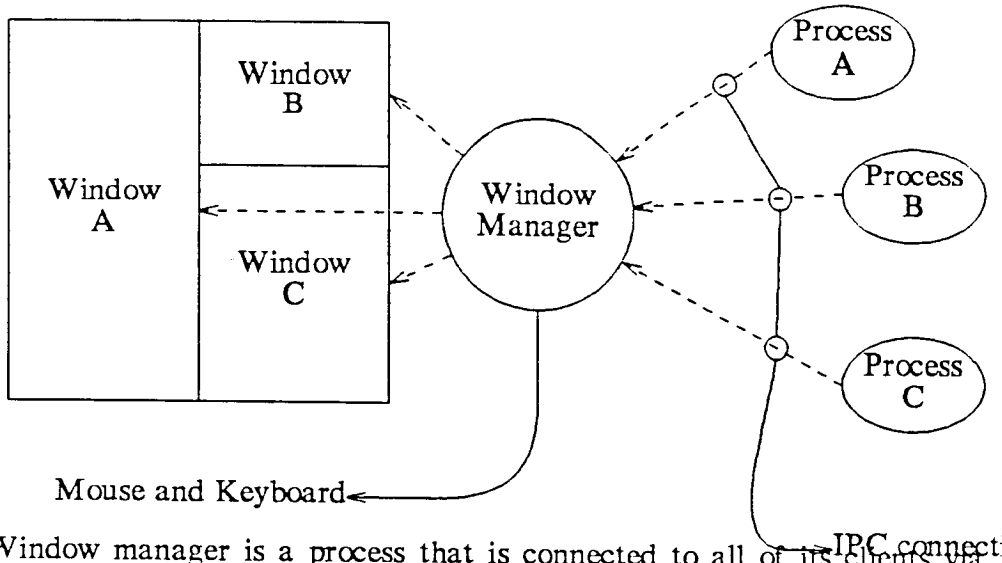
```
                        if (MidpointY)
                                break;
                        pause ();
                }
                wm_ClearWindow ();
                MidpointX /= 2;
                MidpointY /= 2;
                FaceRadius = MidpointX;
                if (MidpointY < FaceRadius)
                        FaceRadius = MidpointY;
                FaceInnerRingRadius = FaceRadius * 19 / 20;
                HourRadius = FaceRadius * 12 / 20;
                HourInner  = FaceRadius *  9 / 20;
                MinRadius  = FaceRadius * 16 / 20;
                MinInner   = FaceRadius *  9 / 20;
                SecRadius  = FaceRadius * 18 / 20;
                SecInner   = FaceRadius *(-6)/ 20;
                for (n = 0; n < 60; n += 5) {
                        wm_MoveTo (FaceInnerRingRadius * angtbl[n].xf
                                        / SCALEANG + Mid-
                                        pointX, FaceInner-
                                        RingRadius * angtbl[n].yf
                                        / SCALEANG + Mid-
                                        pointY);
                        wm_DrawTo (FaceRadius * angtbl[n].xf /
                                        SCALEANG + Mid-
                                        pointX, FaceRadius *
                                        angtbl[n].yf /
                                        SCALEANG + Midpoin-
                                        tY);
                }
        }
        UpdateHand (CurrentTime -> tm_sec,
                        & LastTimeDisplayed.tm_sec, SecRadius,
                        SecInner);
        UpdateHand (CurrentTime -> tm_min,
                        & LastTimeDisplayed.tm_min, MinRadius,
                        MinInner);
        UpdateHand (((CurrentTime -> tm_hour * 5) %60) + Current-
                        Time -> tm_min / 12,
                        & LastTimeDisplayed.tm_hour, HourRa-
                        dius, HourInner);
        wm_MoveTo(MidpointX<<1, MidpointY<<1);
        fflush (winout);
        if (RedrawRequested==0)
                sleep (1);
        else
                RedrawRequested--;
    }
}


UpdateHand (NewTime, OldTime, OuterRadius, InnerRadius)
int    *OldTime; {
```

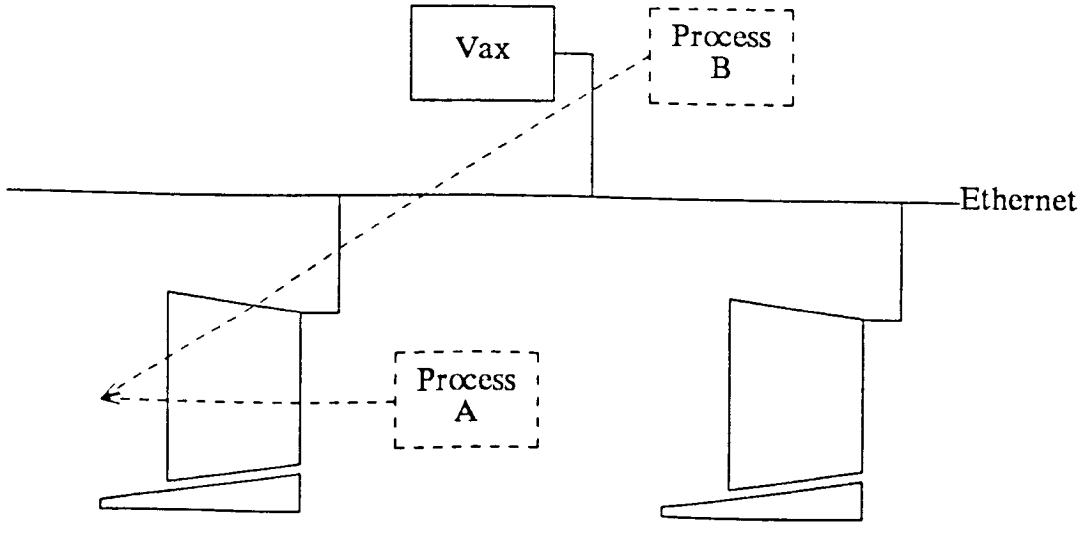
```
        if (*OldTime != NewTime |RedrawRequested) {
                if (RedrawRequested==0 && *OldTime != NewTime) {
                        wm_SetFunction (f_white);
                        DisplayHand(*OldTime, OuterRadius, InnerRadius);
                }
                wm_SetFunction (f_black);
                DisplayHand(NewTime, OuterRadius, InnerRadius);
                *OldTime = NewTime;
        }
}

DisplayHand(Time, OuterRadius, InnerRadius) {
        int OuterX = OuterRadius * angtbl[Time].xf / SCALEANG + MidpointX;
        int OuterY = OuterRadius * angtbl[Time].yf / SCALEANG + MidpointY;
        if (InnerRadius> 0) {
                int LastTime = (Time== 0 ? 59 : Time-1);
                int NextTime = (Time==59 ?  0 : Time+1);
                wm_MoveTo (MidpointX, MidpointY);
                wm_DrawTo (InnerRadius * angtbl[NextTime].xf / SCALEANG
                                + MidpointX, InnerRadius *
                                angtbl[NextTime].yf / SCALEANG +
                                MidpointY);
                wm_DrawTo (OuterX, OuterY);
                wm_MoveTo (MidpointX, MidpointY);
                wm_DrawTo (InnerRadius * angtbl[LastTime].xf / SCALEANG +
                                MidpointX, InnerRadius *
                                angtbl[LastTime].yf / SCALEANG +
                                MidpointY);
        }
        else if (InnerRadius< 0)
                wm_MoveTo (InnerRadius * angtbl[Time].xf / SCALEANG +
                                MidpointX,
                  InnerRadius * angtbl[Time].yf / SCALEANG + MidpointY);
        else
                wm_MoveTo (MidpointX, MidpointY);
        wm_DrawTo (OuterX, OuterY);
}
```

Implementation notes



The Window manager is a process that is connected to all of its clients via socket-to-socket IPC channels. All graphics operations cause messages to be sent between the client and the window manager. The remote procedure call protocol has beed especially crafted for this application. Each procedure call turns into a sequence of bytes. The first is the opcode and the rest (the count is determined by looking up the opcode in a table) are the arguments. Opcodes 0–127 are reserved for the ASCII characters, the rest are in the range 128–255. Many procedure call may be packed into a packet and a procedure call may be split across packet boundaries. If the call doesn't return a value, then it isn't issued immediatly, rather it is just queued up until either the buffer is filled or an explicit fflush(winout) is executed.



Processes on any machine, whether or not they are workstations, can create windows on any display.

Still to be documented:

wm_StdioWindow()
wm_ShowBits(x,y,w,h,b)
wm_DefineRegion(id,x,y,w,h)
wm_SelectRegion(id)
wm_ZapRegions()
wm_SetClipRectangle(x,y,w,h)
wm_WriteToCutBuffer()
wm_ReadFromCutBuffer(n)
wm_RotateCutRing(n)
wm_SaveRegion(id,x,y,w,h)
wm_RestoreRegion(id,x,y)
wm_ForgetRegion(id)
wm_HereIsRegion(id,w,h)
wm_ZoomFrom(x,y,w,h)
wm_LinkRegion(newid,oldid)
wm_NameRegion(id,name)