CMU-ITC-84-023

A Window Manager for Bitmapped Displays and UNIX*

James A. Gosling
David S. H. Rosenthal

Information Technology Center†
Carnegie-Mellon University
Pittsburgh PA 15213

ABSTRACT

A window manager for workstations with bitmapped displays has been developed. It exploits the interprocess communication mechanism of 4.2 Berkeley UNIX, and the DARPA TCP/IP protocols to support remote access to windows. One user-level window manager process runs on each workstation; it tiles the screen(s) 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.

"You will get a better Gorilla effect if
you use as big a piece of paper as possible."
Kunihiko Kasahara, *Creative Origami.*

1. Introduction

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

A window manager for workstations with bitmapped displays has been developed. It exploits the interprocess communication mechanism of 4.2 Berkeley UNIX (4.2BSD), and the DARPA TCP/IP protocols to support remote access to windows. One user-level window manager process runs on each workstation; it tiles the screen(s) 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.

The graphics interface uses pixel coordinates, and primitives such as RasterOp, line, and text. 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.

* UNIX is a Trademark of AT&T Bell Labs.
† The Information Technology Center is funded by IBM.

James Gosling's present address is Sun Microsystems Inc., 2550 Garcia Ave., Mountain View, CA 94043

Parts of this work were covered by presentations at the UniForum (Washington DC, January 1984) and Computer Graphics '84 (Anaheim CA, May 1984) conferences.

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 direct access from user processes to the hardware. Client processes using the window manager to perform bitmap operations are *portable*, in the sense that they will run on any 4.2BSD system on the network, and *device-independent*, in that they can have no knowledge of the hardware being managed on their behalf.

1.1. Development Context

The charter of the Information Technology Center (ITC) is to introduce a network of many thousands of workstations to the Carnegie-Mellon campus. All students, staff, and faculty will eventually have powerful personal computers with bitmapped displays. Major applications are expected to include electronic mail, text processing, and educational software.

The software for this project is being developed on a network of about 100 Sun Microsystems workstations; the workstations eventually deployed are expected to offer similar performance and display capabilities, and to run the same 4.2BSD operating system. Since all the ITC software will have to be ported to these machines, a major consideration in the design of the window manager was device independence and portability.

The window manager became operational in November 1983, and has been used within the ITC as the basic user interface tool ever since. It will be released to application developers on the campus in October 1984, and will be licenced to outside users shortly thereafter.

1.2. User's View

The user of a workstation running the window manager sees one or more screens *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 identification 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.

<Figure 1 –Screen Layout>

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 form a hierarchy: 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.

<Figure 2 –Menu Example>

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 existing 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.

2. Client's View

2.1. Design Principles

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

A simple interface to the window manager forms the base for a series of layered interfaces. Clients can choose an interface at an appropriate level, for example GKS@Cite(GKSReport), and it can be implemented as a subroutine library. Further, a simple window manager is likely to be fast. Inefficiency in the primary user interface tool is unlikely to be tolerated.

Viewing resource requests as hints has three 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 must take device independence seriously; they cannot predict the properties of their "virtual device" because they will change at run-time.

2.2. Resource Model

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. These values may change at any time.

The pixels in this array may be treated in one of two ways:

- As containing one of a number of colours. The client may define a colour using a string name (e.g. *LimeGreen*), or an RGB triple, and the result will be a handle that may be used to select it. The window manager will use its best efforts to supply a suitable approximation.
- As containing one of a contiguous range of indices into the colour map. The client may hint as to the number of indices it requires, and inquire as to how many it has actually been given. This value may change at any time. It can manipulate the colour map entries corresponding to its current set of indices.

Characters may be drawn into this pixel space in any of several fonts. A client may define a font using a string name (e.g. *TimesRoman12*), and the result will be a handle that may be used to select it. The window manager will use its best efforts to supply a suitable approximation. The client may inquire about the bounding box of any character in the font. The size and shape may change at any time.

Whenever the window manager changes the allocation of resources so as to affect a particular client, that client is notified. It is expected to inquire what the new allocation is and re-draw the image in its window to suit.

2.3. Control Functions

Among the functions used to pass control information between the client and the window manager are:

w = DefineWindow(h)

Creates a window *w* on host *h*.

SelectWindow(w)

Makes *w* the current window.

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 creates a communication channel to the window manager on that host, and returns a handle giving access to the window's resources. If the window is on the same host, the client will eventually be able to choose to do graphics output via memory-mapped access to the device registers or via the communication channel to the window manager. The choice will be made by loading a different (and bigger) library; it will not be visible at the interface.

Many 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.

Notification that resource allocations have changed (for example, that a window has changed size) 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 re-sized (or having other resources changed). 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.

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).

2.4. Primitive Attributes

Among the functions setting the attributes applied to graphic output are:

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 colour whose name is the string *n*.

SelectColor(c)

Selects the defined colour *c* for subsequent drawing.

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. Since the user may re-position a window from one display to another, the values in the font structure may change dynamically. They are obtained from the window manager, and are updated when the client is notified of a font change.

2.5. Output Primitives

The client draws using a functional interface. A "painting" model of graphics output is used@Cite(WarnockWyatt); the full generality of RasterOp@Cite(NewmanSproull) is not made available to clients as it does not generalise well to colour displays. Examples of the primitives available are:

DrawTo(x,y)
MoveTo(x,y)

Moves the current position to (*x*,*y*), with and without drawing a line on the way.

CopyRaster(sy,sy,dx,dy,w,h)

Copies the rectangle with upper left corner (*sx*,*sy*) to that with upper left corner (*dx*,*dy*), both are (*w*,*h*) wide and high.

PaintRaster(dx,dy,w,h)

Paints the destination rectangle with the selected colour.

FillTrapezoid(x1,y1,w1,x2,y2,w2,f,c)

Fills the specified trapezoid with the character *c* from the font *f*.

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 routines for drawing strings relative to a positioning parameter. These apply only to the selected window. For example:

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

draws the string *Center* centered vertically and horizontally in the current window. Use of *DrawString* obviates the need to remember potentially variable parameters from the font structures.

2.6. Input Functions

Characters typed on the keyboard are routed by the window manager to the process owning the window pointed to by the cursor. They are simply written to the IPC connection for that window; the client process can read them from the corresponding file descriptor in the normal way.

AddMenu(s, r)

Adds the string *s* as an item in the pop-up menu of the selected window. The response string is *r*.

When a menu item is selected, the effect is as if the response string had been typed. In this way, menus can easily be added to existing programs.

SetMouseInterest(mask)

Informs the window manager of the type of mouse events the client is interested in.

SawMouse(& action, & x, & y)

Sets *action* to the event code, and *x* and *y* to the position in the window.

Whenever an "interesting" mouse event occurs, the un-typeable character *MouseInputToken* appears in the input stream. The client should then call *SawMouse* to decode the following few characters. Interest masks and actions are composed by or-ing together values including *LeftButton*, *DownMovement*, and *UpTransition*.

3. Implementation

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 4.2BSD sockets.

<Figure 3 –Process Structure>

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 4.2BSD system calls. No kernel changes were needed.

3.1. RPC Implementation

A *socket*, as defined by 4.2BSD, 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, 4.2BSD 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.

The RPC protocol supports C-style parameter passing, with at most one variable-length argument (typically a string). Functions may return results directly, or they may be stored through pointer arguments. The common operations are output primitives and attribute selections, which do not return values. These are accumulated in *stdio* buffers until explicitly flushed, or until a result is required. Typical interactions between the window manager and the client consist of a single message from the client containing many operations.

3.2. Window Creation

When the client requests the creation of a window a communication path is set up to the window manager. The window manager creates a structure in its address space describing the properties of the window, but does not actually create it until a request is received that requires its existence. Thus, the window creation heuristics can use any information sent to the window manager in the meantime.

The window creation heuristics use 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. A system of equations was relaxed to minimise the energy in the compression of the springs. This was very uncomfortable to use; it almost always completely rearranged all windows every time a new window was created. The present heuristics 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 are determined by minimising an error function that attempts to balance areas and preserve aspect ratios.

3.3. Font Support

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 receive 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 independent 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.

4. Applications

One of our applications is an editor@Cite(GoslingProtext), 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 editor is fast enough to be used as a general user interface tool, in effect a replacement for the Unix "teletype driver". Users generally use the Unix shell via a typescript manager implemented from the same code. It keeps a complete transcript of their session, permits scolling, string search, and text to be cut and pasted[†].

Another application is a diagram editor, implementing primitives such as lines, arcs, ovals, boxes, arrows, and text. The primitives may be compsed into symbols; both primtives and symbols may be placed in the diagram and linked by connectivity constriants at "magic dots". Objects stay connected even though the dots are moved. The window manager supports this adequately except that dragging objects around the screen is too slow (the mouse event has to pass through the window manager on its way to the diagram editor, and the response has to pass through the window manager on the way back. Three processes must be scheduled).

++++++++++++++++

† The window manager provides a ring of cut buffers, permitting text cut from one window to be pasted into another, even across machine boundaries.

5. Assessment

5.1. User's View

It is usual for window managers to permit 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 to use heuristics when allocating space for new windows, and to 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. A common complaint from users is that the layout process is unpredictable. We are experimenting with Cedar-style fixed-width columns of windows as an alternative. Clients are, of course, unaware of the layout policies being followed by the window manager.

5.2. Client's View

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. 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.

It can be claimed that the requirement to re-draw the window 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.

5.3. Device Independence

Device independence is essential; the campus network will be an open system, and applications must operate on incompatible workstations. All access to the display hardware is mediated by the window manager, effectively insulating applications from its peculiarities@cite(CahnYen). The window manager currently has device drivers for three Sun displays, the earlier 1024-by-800 monochrome, the more recent 1152-by-900 monochrome, and the 640-by-480 colour. Porting it to future workstations will involve writing new device drivers; applications will be as unaware of the differences as they are between the existing three display types.

5.4. Performance

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, requiring 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