5 SunDew - A Distributed and Extensible Window System

James Gosling

5.1 INTRODUCTION

SunDew is a distributed, extensible window system that is currently being developed at SUN. It has arisen out of an effort to step back and examine various window system issues without the usual product development constraints. It should really be viewed as speculative research into the *right* way to build a window system. We started out by looking at a number of window systems and clients of window systems, and came up with a set of goals. From those goals, and a little bit of inspiration, we came up with a design.

5.2 GOALS

A clean programmer interface: simple things should be simple to do, and hard things, such as changing the shape of the cursor, should not require taking pliers to the internals of the beast. There should be a smooth slope from what is needed to do easy things, up to what is needed to do hard things. This implies a conceptual organization of coordinated, independent components that can be layered. This also enables being able to improve or replace various parts of the system with minimal impact on the other components or clients.

Similarly, the program interface probably should be procedural, rather than simply exposing a data structure that the client then interrogates or modifies. This is important for portability, as well as hiding implementation details, thereby making it easier for subsequent changes or enhancements not to render existing code incompatible.

Retained windows: a clean programmer interface should completely hide window damage from the programmer. His model of a window should be just that it is a surface on which he can write, and that it persists. All overlap issues should be completely hidden from the client. I believe that the amount of extra storage required to maintain the hidden bitmaps on a black and white display is negligible - based on the observation that people generally do not stack windows very deeply. The situation is somewhat different with colour, but there are games to be played. Retained windows is one way of hiding window damage, but we do not want to commit to a particular solution to this problem at this time.

Flexibility: users need to attach devices, change menu behaviours and generally modify almost all components of the system. For example, the menu package ought to be independent of the particular format or contents of the menu, thereby allowing the user to develop his own idioms without having to reimplement the entire system.

Part of flexibility is device independence: SUN provides a spectrum of display devices to which clients need consistent and transparent interfaces. This leads directly to portability, which we also need to achieve.

Users should be able to make various tradeoffs differently than in the standard system, because of either particular hardware or performance requirements. For example, if the system provides retained windows because we believe that the cost in terms of memory usage is worth the performance improvements, a user should be able to make this tradeoff differently, for example if he has less memory.

This extreme flexibility might appear to be at odds with having a clean, simple, well-abstracted programmer interface, but we do not believe that it is.

Remote access to windows: in the kind of distributed networked environment that SUN promotes, it is natural to want to be able to access windows on another machine as naturally as the NFS promises to support accessing remote files. We believe that this will fall out of any reasonably designed system.

Powerful graphical primitives: the primitives that the Macintosh provides should be considered as a lower bound. Curves and colour need to be well integrated. Attention should also be paid to what CGI [30], GKS [28], CORE [24] and PHIGS [6] need. A consequence of an emphasis on power and flexibility is the ability to emulate other window systems, eg it would be very valuable to be able to provide an emulation of the Macintosh toolbox.

Exploit the hardware: in particular, none of the systems mentioned above deal well with colour. In the future, colour is going to play an even larger part in display design. One can view black and white as a temporary technological stopgap, just as happened with television. Besides, SUN makes some pretty good colour displays, so the window system should exploit them. One implication of this is that the font file format must completely hide the details of the representation of characters, since we might eventually want to support antialiased text, and even illuminated monastic typefaces.

Perform well: the performance of the current window system should be considered as the minimum acceptable level. Performance in the common cases is especially critical. The new system should perform faster than the current system on such common operations as repainting and scrolling of text.

5.3 DESIGN SKETCH

The work on a language called PostScript [1] by John Warnock and Charles Geschke at Adobe Systems provided a key inspiration for a path to a solution that meets these goals. PostScript is a Forth-like language, but has data types such as integers, reals, canvases, dictionaries and arrays.

Inter process communication is usually accomplished by sending messages from one process to another via some communication medium. They usually contain a stream of commands and parameters. One can view these streams of commands as a program in a very simple language. What happens if this simple language is extended to being Turing-equivalent? Now, programs do not communicate by sending messages back and forth, they communicate by sending programs which are elaborated by the receiver. This has interesting implications on data compression, performance and flexibility.

What Warnock and Geschke were trying to do was communicate with a printer. They transmit programs in the PostScript language to the printer which are elaborated by a processor in the printer, and this elaboration causes an image to appear on the page. The ability to define a function allows the extension and alteration of the capabilities of the printer.

This idea has very powerful implications within the context of window systems: it provides a graceful way to make the system much more flexible, and it provides some interesting solutions to performance and synchronization problems. SunDew contains a complete implementation of PostScript. The messages that client programs send to SunDew are really PostScript programs.

Two pieces of work were done at SUN which provide other key components of the solution to the imaging problems. One is Vaughan Pratt's Conix [53], a package for quickly manipulating curve bounded regions, and the other is Craig Taylor's Pixscene [63], a package for performing graphics operations in overlapped layers of bitmaps.

Out of these goals and pieces grew a design, which will be sketched here. The window system is considered in four parts. The imaging model, window management, user interaction, and client interaction. The imaging model refers to the capabilities of the graphic system - the manipulation of the contents of a window. Window management refers to the manipulation of windows as objects themselves. User interaction refers to the way a user at a workstation will interact with the window system: how keystrokes and mouse actions will be handled. Client interaction refers to the way in which clients (programs) will interact with the window system: how programs make requests to the window system.

What is usually thought of as the *user interface* of the window system is explicitly outside the design of the window system. *User interface* includes such things as how menu title bars are drawn and what the desktop background looks like and whether or not the user can stretch a window by clicking the left button in the upper right hand corner of the window outline. All these issues are addressed by implementing appropriate procedures in the PostScript.

5.3.1 Imaging

Imaging is based on the stencil/paint model, essentially as it appears in Cedar Graphics [65] and PostScript. A stencil is an outline specified by an infinitely thin boundary that is piecewise composed of spline curves in a non-integer coordinate space and which may be self-intersecting. Paint is some pure colour or texture - even another image - that may be applied to the drawing surface. Paint is always passed through a stencil before being applied to the drawing surface, just like silkscreening. This is the *total* model: lines and characters can be defined using stencils. Lines are done as narrow stencils - underneath it all, it is not really done this way: special cases are exploited wherever possible.

One can think of a stencil as a clipping region. Stencils may be composed by union, intersection and difference to create new stencils.

The Macintosh graphics system, QuickDraw [18], can be easily cast in this framework: their GrafPort is simply a paint box, and all of the shapes that can be drawn are subsets of the capabilities of stencils. Many other 2D standards fit in easily as well, and the extension of the model to 3D is relatively straightforward.

Vaughan Pratt's work on conic splines presents us with an opportunity actually to realize this model and achieve good performance.

Almost all imaging functions are provided not by the window manager but by PostScript processes. For example, if one wanted rounded rather than square corners on a menu package, this could be done by modifying a PostScript procedure. Scope rules determine whether modifications are global or local.

5.3.2 Window Management

Pixscene [63] makes windows cheap and easy to create. If a client wants to create a window quickly as a pop-up, the window system will handle it. It very efficiently makes sure that each window is always complete: overlap is not visible to the client.

Even though windows behave as though the image is retained, clients will have to deal with requests to redraw from the window system. This will generally only happen when a window is stretched, in which case the client probably has no use for the old image, anyway.

Pixscene is based on a shape algebra package. The ability, provided by Conix, to do algebra very rapidly on curves should make non-rectangular windows perform well.

5.3.3 User Interaction - Input

The key word in the design of the user interaction facilities is *flexibility*. Almost anything done by the window system preempts a decision about user interaction that a client might want to decide differently. The window system therefore defines almost nothing concrete. It is just a loose collection of facilities bound together by the extension mechanism.

Each possible input action is an *event*. Events are a general notion that includes buttons going up and down (where buttons can be on keyboards, mice, tablets, or whatever else) and locator motion.

Events are distinguished by where they occur, what happened, and to what. The objects spoken about here are physical, they are the things that a person can manipulate. An example of an event is the 'E' key going down while window 3 is current. This might trigger the transmission of the ASCII code for 'E' to the process that created the window. These bindings between events and actions are very loose, they are easy to change.

The actions to be executed when an event occurs can be specified in a general way, via PostScript. The triggering of an action by the striking of the 'E' key in the previous example invokes a PostScript routine which is responsible for deciding what to do with it. It can do something as simple as sending it in a message to a Unix process, or as complicated as inserting it into a locally maintained document. PostScript procedures control much more than just the interpretation of keystrokes: they can be involved in cursor tracking, constructing the borders around windows, doing window layout, and implementing menus.

Synchronization of input events: we believe that it is necessary to synchronize input events within a user process, and to a certain extent across user processes. For example, the user ought to be able to invoke an operation that causes a window on top to disappear, then begin typing, and be confident about the identity of the recipient of the keystrokes. By having a centralized arbitration point, many of these problems disappear.

5.3.4 Client Interaction

Client interaction with the window system can be broken down into three layers: messages that get passed between the window system and the client, the programs that are contained in the messages, and a procedural interface to program construction. This is one of the key unique points about this window system: clients send *programs* to the window system whose elaboration causes graphic operations to appear. The client program does not send graphic operations directly. There is, of course, a procedural interface to program construction that blurs the distinction.

This approach allows the client to redefine the protocol to compress messages passed to the window system. For example, if the client wishes to draw a grid it can download a procedure which iteratively draws the lines, generating their coordinates as a function of the loop index. This is a substantial compression over transmitting a large set of lines, even if the grid is drawn only once. There are other performance advantages: message-passing interactions are relatively slow and downloading a procedure that will respond locally eliminates this overhead. For example, if a client program needs rubber band lines, then rather than have the window system send a message to it each time the mouse moves, it can download a procedure that will track the mouse locally.

There are a number of features of PostScript that make it attractive. It is very simple, and it is complete. Its simplicity makes it fast to translate and interpret - this can almost be done as quickly as packets can be disassembled in a more traditional IPC environment. Adobe did a very good job of designing a set of primitives and data structures that fit together well. Its chief drawback is that it can be hard for people to understand; the postfix notation is well-suited to consumption and generation by programs, but humans find it obscure.

It is important to think about the client programmer's model of what the window system does. We expect there to be two levels of model. The first completely hides the existence of PostScript with a veneer of procedure calls that construct and transmit fragments of PostScript programs. The second exposes PostScript. Beyond a certain level of functionality, learning PostScript is inevitable: it can be pushed off by making the veneer more comprehensive, but this just makes the eventual leap harder.

In order to give a flavour of PostScript, a small example is given in conclusion.

A PostScript Example

The PostScript code to generate the figure shown in **Figure 5.1** is:

```
clippath pathbbox newpath
2 div exch 2 div exch translate
250 250 scale 90 rotate
25 (0 .9 moveto 0 0 1 90   -90 arc
        0 0 .9 -90 90 arcn fill
        .88 .88 scale 22.5 rotate) repeat
```

The first three lines set up the environment, centre the image and set up the transformation matrix. Then the coordinate system is changed by scaling down .88 in both directions and rotating 22.5. This is repeated 25 times.

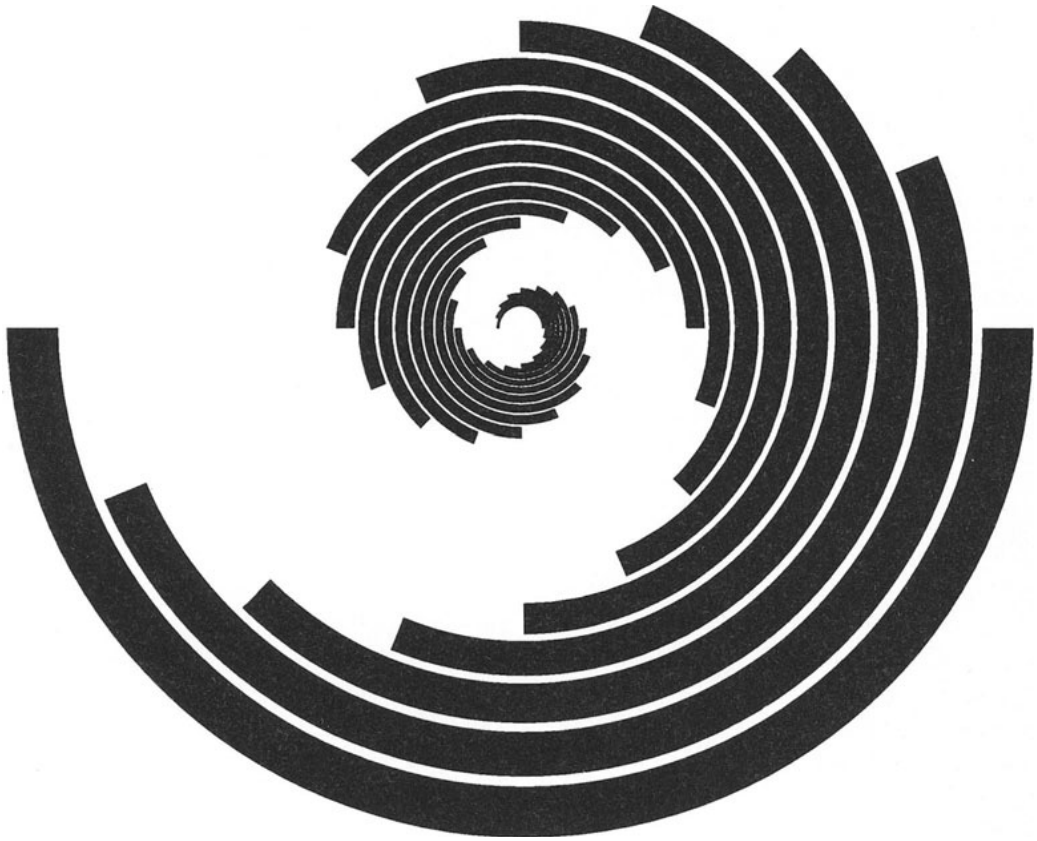

Figure 5.1

5.4 DISCUSSION

Chairman - George Coulouris

Williams: What is the scope of the devices you are considering? I don't suppose you intend running the window manager on a graph plotter.

Gosling: The crudest display we are willing to accept is 1 bit per pixel black and white but we also support 8 or 24 bits per pixel colour or 4 bits per pixel black and white.

Williams: Essentially bitmap raster devices.

Gosling: Right - although when you get to greyscale devices, things stop behaving in a model that is comfortably compatible with RasterOp. You have got to be able to deal with antialiasing, such things as subpixel positioning begin to make sense. It makes sense to draw a character midway between two pixels because you can use antialiasing to shift the character over by subpixel amounts.

Bono: When you use the word PostScript, do you mean literally that PostScript is in some way your virtual machine instruction set? It has not been extended or generalized?

Gosling: It has been extended. It is a superset. We are committed to implementing everything in the PostScript telephone book that makes sense. They have a few commands that are particular to their storage allocation and to the fact that they are going to a printer and these are not implemented. We have imported some things that are peculiar to a display and a small number of incompatible changes to the language have been made which we spent a long time talking to the people at Adobe about to make sure that was reasonable. In particular, we added garbage collection and lightweight processes. There are very tiny ways in which the semantics of PostScript's memory allocation strategy shows through in the printer version because they have a quick and dirty storage allocation mechanism and that wasn't really useful in this system.

Bono: The virtual machine was not virtual enough.

Gosling: Right. When we made the generalization to multiple processes, their storage allocation mechanism just completely broke and so we had to use garbage collection instead and that necessitated some small semantic changes but they are not things you are likely to see. All of the important things such as how you specify a curve, whether you can render an image rotated 37 degrees, all of that is there or intended to be there.

Hopgood: How do you handle input?

Gosling: Input is also handled completely within PostScript. There are data objects which can provide you with connections to the input devices and what comes along are streams of events and these events can be sent to PostScript processes. A PostScript process can register its interest in an event and specify which canvas (a data object on which a client can draw) and what the region within the canvas is (and that region is specified by a path which is one of these arbitrarily curve-bounded regions) so you can grab events that just cover one circle, for example. In the registration of interest is the event that you are interested in and also a magic tag which is passed in and not interpreted by PostScript, but can be used by the application that handles the event. So you can have processes all over the place handling input events for different windows. There are strong synchronization guarantees for the delivery of events even among multiple processes. There is nothing at all specified about what the protocol is that the client program sees. The idea being that these PostScript processes are responsible for providing whatever the application wants to see. So one set of protocol conversion procedures that you can provide are ones that simply emulate the keyboard and all you will ever get is keyboard events and you will never see the mouse. Quite often mouse events can be handled within PostScript processes for things like moving a window.

Sweetman: How do windows relate to canvases?

Gosling: I did not use the word window because its overloaded with all kinds of semantics. Does it have a border? All that a canvas is, is a thing on which you can draw. It is not even rectangular.

Sweetman: Do you see canvases on your display?

Gosling: Yes you can. A canvas is a thing on which you can draw. It might be visible on some display and it might not. If it is visible, you can specify its position in a 2½D coordinate system. It is opaque but you can get a transparent effect if you want something to show through a window, by cutting a hole in the window.

Williams: You say clients have to accept redraw commands. Is there any indication as to how soon they are supposed to do them?

Gosling: The client can ignore the request if it wants. The screen image will look a little funny. Or it can wait half an hour - it does not affect the integrity of the screen. It will affect the integrity of its window but nothing else because all the canvases are maintained in apparent isolation from each other.

Williams: Will the visual image just contain what was there before?

Gosling: You can do that, if you wish. For most applications it does not make any sense to retain the old bitmaps so you might as well just blow them away and replace them by whatever is most convenient - which is probably what was there before. There are some times when you would like to retain the old bits - that is there as an option. I really want to make it difficult for people to exercise that option as it is a very tasteless thing to do in general.

Bono: I hope I am not being overly opaque about this but it seems like I have a sense of déjà vu in that it's a lot better way of doing what people did with display lists, where you sent things down to Vector Generals or whatever. They were little programs and there was some grouping structure but everything is done so much nicer now in the sense of having complete programs and a better set of primitives etc. Am I missing something from the model or is that really what is going on?

Gosling: One of the things that tended to characterize all the display list languages was that they tended to be tailored very much towards what the hardware could do. PostScript tries to stand back and say "I don't want to know what the hardware can do, I want to know what makes sense for the user".

Bono: What I would like people to do when they do look at standards, instead of ignoring them, is to look at, for example, the CGI work; this is a set of functions which some people claim is a good instruction set (forget the syntax for the moment, just look at the functionality). We would

like to get some comment on whether it is a good set of functions and that is what you should be looking at rather than ignore the standards. All of this could be fitted in a standards context if we get it right.

Gosling: One could easily use the CGI graphics model for this instead.

Teitelman: The innovation here is not that we are using PostScript. The reason we chose PostScript is due to a lot of historical connections and proximity to the people who are doing it.

The interesting thing is that all these processes look as though they are executing in one of those old single address environments. It is a single user process that James has been able to implement lightweight processes in. You don't have lightweight processes in Unix systems, which you really need to implement realistic user interfaces.

Gosling: There is really nothing new here. It's just putting it together in a different way.

Rosenthal: It is worth bringing out a number of points about this style of window manager. There are some real limitations, though I think this is the way we should go.

(1) Some help from the kernel is needed. This is easy in 4.2 which has a sensible IPC. It could be done in System V through shared resources etc.

(2) A reliable signal mechanism is essential. The window manager has to stay up. It is the only routine for talking to the system and hence must stay alive. We have 100 systems running and only experience 1 - 2 crashes per week. This is good by Unix standards!

(3) Applications cannot read pixels back - this is just too slow.

(4) There **must** be a way to make the client think it has a real terminal, such as the 4.2BSD PTY device, otherwise all old programs misbehave.

(5) There must be a mechanism for downloading application code into the window manager process, for example, rubber band line mouse tracking is done by a user level process.

(6) There must be off-screen space in the window manager.

Examples of systems built in this way are:

BLIT [50];
Andrew (CMU)(see Chapter 13);
VGTS (Stanford) [36];
X (MIT).

The BLIT system is successful - people wrote programs for it. I believe it is the only successful window manager for a Unix system.

Myers: Maybe that is because BLIT doesn't run a Unix operating system.

Rosenthal: Overall, it is a Unix software environment.

Williams: We should bear in mind that Rob Pike says that to do anything sensible using the BLIT, you need to write two processes which communicate: one running in the Unix host and one running in the BLIT, and this must be harder to do than, say, writing two programs which don't communicate.