Window System Design: If I had it to do over again in 2002.

James Gosling
December 9, 2002

In the deep dark past I have been involved in building window systems. I did the original design and implementation of both the Andrew and NeWS window systems. Both of which predated X11. They shared with X11 the architectural feature of being networked: clients sent messages to the server over TCP connections. I occasionally get asked "if you had to do it over again, what would you do? Would you do the same thing". The answer is a strong no. It's now 20 years later, and the technological landscape is totally different. So here is what I would do. But first…

Background

The term "window system" is somewhat loose. It generally refers to the mechanism by which applications share access to the screen(s), keyboard and mouse. Beyond this it generally contains facilities for inter-application messages such as support for cut-and-paste, and drag-and-drop. It also often contains support for the decorations surrounding windows that provide the user interface for resizing, opening and closing windows; although in some systems this has been left up to the application. Sometimes the window system provides higher level abstractions like menus.

History

When a system is designed, there are always tradeoffs made that reflect the technology of the day. In the case of Andrew and NeWS, these tradeoffs were based on the state of the art 15 to 20 years ago (this probably applies to X11 too, but I wasn't involved in the design analysis behind it). There were a number of things that were very different between then and now.

1) The most significant is the relative performance of graphics rendering and network communication. Back then, rendering was relatively slow. The overhead of network communication was significantly overshadowed by the overhead of rendering.

2) Back then, there were no shared libraries. This seems odd, looked back at from today, but back then no version of Unix had the ability to have a library like libc or OpenGL that was shared between processes. All applications had to be "statically linked". There was a primitive segment

sharing facility that allowed one segment per process to be shared, that was at the beginning of the address space; but it wasn't powerful enough for this purpose.

3) Putting large things, like windowing libraries, into the kernel is generally a bad idea. It has a significant negative impact on the reliability and testability of the system.
4) When hardware acceleration was available, it generally had no interlocking mechanisms for arbitrating amongst independent threads that were trying to use it. This generally meant that either the accelerator was permanently allocated to a thread (very common, since acceleration was normally 3D hardware used exclusively for CAD), or there was an software interlock mechanism that added some cost to each operation.

So, given these, where do you put all of the code that is involved in the window system – including the graphics rendering library? Remember that rendering libraries tended to be large, since hardware acceleration was almost non-existent.

They couldn't be in each user process, since without being shared, they would take up an unacceptable amount of RAM. So the only way to get one copy of the code, and have it outside of the kernel, was to have it in one process, and to have applications communicate with this "window server".

What's different

But today, while putting large amounts of code into the kernel is still a bad idea, rendering performance has improved dramatically, and most operating systems have shared libraries. The increase in rendering performance has outstripped Moore's law, which in turn has outstripped the increase in generally available bandwidth, making the overhead of shipping requests through the network an unacceptable burden.

High performance 3D rendering hardware has become so common that it is actually difficult to buy a computer that doesn't have any. And an important feature of most accelerators is that they have facilities to efficiently handle multithreaded access. It has become a standard part of a computer's architecture, alongside integer arithmetic, floating point arithmetic, and networking.

The level of aspiration of applications has increased dramatically. The 3D hardware is there because it is used. Even old-school applications, like word processors, that one would think of as just needing simple 2D graphics are using acceleration for such things as image scaling.

It's been interesting to watch the evolution in the way applications use X11. It has become standard to sidestep the servers rendering and use the direct screen access extensions and libraries like libart.

Principles for a New Design

1) Acceleration is normal, not rare.
2) Shared libraries are everywhere.
3) The code path from the application to the accelerator needs to be as short as possible.

The design

Given these, the basic outline of the design falls out fairly naturally. I would make the "window system" so minimal that it is almost non-existent. Each graphical application gets direct access to the hardware, and a window is nothing more than a clipping list and an (x,y) translation. I would build a "device driver" that did nothing more than manage the clipping lists and hand out graphic device ports. This might actually be best done at user level, rather than a device driver, using shared memory and semaphores.

There are a variety of "hairy bits" that make this more complicated:

It doesn't just maintain clipping lists. It maintains the "true shape" of each window, and a stacking order. The windows clip is derived from these by subtracting from the clip for a window the shapes of all of those above it. Whenever the shape or stacking order of a window is changed, clip lists get updated. If an application has its clip list changed I would notify it via a message on the mouse/keyboard event queue. Until the application acknowledges the clip change, the old window shape has to be considered as being continuously damaged by the application.

It has to handle resource allocation within the accelerator, including texture space and rendering ports.

Some other device drivers would be affected:

1) The mouse driver would need to have an association between clipping lists and processes to deliver events to. It is the natural place for one small piece of UI policy: that once a mouse button goes down, all following events get delivered to the same process until all buttons are up.
2) The keyboard driver would need to have a notion of the "current focus" and send keystrokes only to that process (==file descriptor, from the drivers point of view). And an IOCTL to request the focus, and some pseudo keystroke events to indicate focus gained and lost.
3) It is quite likely that the keyboard and mouse driver should be merged at the higher levels so that applications only

have one file to read events from. This also straightens out temporal ordering hazards between the mouse and the keyboard.

Almost everything else would be done at user level, in the user process: rendering, window borders, and the hard parts of cut/paste/drag/drop. It also needs to have its own mouse/keyboard distribution since the kernel only does coarse grained distribution based on window shape. The kernel knows nothing of the details within a window.

I wouldn't use signals for anything. Everything would go through a unified message queue (along with mouse and keyboard events).

Implications

This allows a wide range of rendering libraries to be supported. The window system knows nothing of rendering and imposes no preconceived notions on it. I wouldn't expect applications to deal directly with rendering, rather I'd expect a small number of rendering libraries to be written. The three most likely candidates are OpenGL, something that does roughly the PostScript rendering model, and something that does roughly the X11 rendering model (it may also have an Xlib compatible API).

Window borders Do them in the application library!

Painting policy (double buffering) These days it's becoming required to implement double buffering. For anyone who's ever used OSX, the experience is totally addicting and quickly become a non-negotiable feature (along with anti-aliased fonts!). One of the nice things about this design is that it allows for either use of double-buffering hardware or OSX-like rendering offscreen and compositing into the framebuffer. It's up to the rendering library. This does introduce a tricky bit of coordination that will have to be worked out: how to manage the compositing & buffer flipping, particularly in the face of video sync issues.

X11 compatibility There are two cases: for local X11 apps, the best solution is an Xlib compliant rendering library that does everything locally. I would not push the compatibility library too far: in particular, all the X facilities for supporting window managers I would just flush. The remote case is somewhat harder. One could just port an X11 server to this environment, much as it has been ported to OSX. This would be pretty heavyweight, although quick and easy to put together. I think that a more viable solution in the long run would be to replace the X protocol with a very simple pixel copying protocol that uses the user-level rendering libraries in the application to render to a local image buffer, then copies the pixels over the net in something that looks vaguely like a video stream.

There are a variety of compression hacks that make this surprisingly efficient – this is essentially what the SunRay product does. Some analysis has been done that shows that this uses essentially the same bandwidth as the X protocol, if done well. It has the advantages of being both a lighter weight solution and allowing rendering APIs other than Xlib to be used remotely.

The result

Would be a system that is both lightweight and fast. Everything could move at the speed of a finely tuned video game. Advances in rendering pipelines and library design would be easy to accommodate. This window system design isn't particularly radical: it's more just pointing out that this is the way that X is going already, given the increasing predominance of application-side rendering libraries. Once you accept that fact and admit that it's actually the right way to go, the design falls out, simply by stripping away legacy stuff that isn't needed any more.