

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-04/msg00087.html>

- *From:* "Beth" <BethStone21@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Sat, 02 Apr 2005 15:49:35 GMT
-

Johannes Kroll wrote:

> Beth wrote:

>> Yes; "Double buffering"...

>> When using DirectX in "full screen mode", then this is, in fact,

>> exactly what happens...

>>

>> Simply, they can't use "double buffering" in the windowed mode

>> because, originally, video cards / system RAM could NOT afford to have

>> a second screen (too much RAM)...and, once that decision has been

>> taken, then they have to maintain it for "backwards compatibility"...

>>

>> Also, "double buffering" in games often uses a "flip" method for fast

>> animation (the "back buffer" and "front buffer" are simply

>> "swapped")...the problem is that, in windowed mode, you'd then either

>> have to ask all the windows to "repaint", in order to re-build all the

>> windows in the new buffer..._OR_, alternatively, when the screen if

>> "flipped", the "front buffer" has to be _COPIED_ to the "back

>> buffer"...

>

> I know, you can do the same with the VGA registers, "linear display

> address low/high" or so it was called.

Yeah, that's the one...

But, for "windowed mode", then "flip" wouldn't really work – have to send "paint messages" to all applications all the time to stitch the graphics back together – unless you 1. Used the "display list" concept (see? There's method in the madness ;) 2. Copied the "front buffer" to the "back buffer" all the time...

> GL actually "copies" everything, but it's done by the graphics card, so

> it's fast, and you don't need to redraw everything. GL is unusable

> anyway if it's not hardware accelerated.

Yeah, I know...the OpenGL screen savers on Linux were stupidly slow until I installed the proper driver, then it zooms around the screen...

> If you don't have hw acceleration (e. g. with SVGAlib...) you can put

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

- > the backbuffer in system ram and update only the modified screen
- > regions. I tested this and it works relatively fast even on a 200MHz
- > system... :)

Actually, yes...unfortunately, IBM never wired up the memory-to-memory transfer with the standard DMA chip, apparently...which is annoying because I was trying to do something like that before (in DOS and VGA but same principle), as it would have been great to keep the backbuffer in RAM for writing the pixels and then using the DMA chip to transfer it, in parallel, to the video memory ("front buffer") for display...without that "copying" requiring the CPU, that could work on the next frame (perhaps "triple buffering" to avoid "tearing" of any sort)...but, nope, the DMA chip is capable of this in general (you need to use two DMA channels that you "connect" together...one reads, the other writes...it's in the chip's manual, anyway :)...it's just that it was never wired up that way inside the PC and only works when "I/O" is involved on one or other end (at least, checking on websites to see if anyone else had done this, a few had tried but no-one was successful, even though the DMA chip's manual says it's capable of this ability, which strongly suggests that it wasn't wired up in the PC correctly to make use of this...a pity because the DMA being able to do memory-to-memory copies without CPU involvement would, to me, seem far more generally useful as a "blitter"...would have been the first "hardware acceleration"...that's a bit of a missed opportunity there, really...although, it's all "academic" today, really, what with accelerated 3D cards and the AGP bus and such :)...)

- > Anyway, the "persistent display scheme" (didn't know it was called like that) isn't hw dependent.

Exactly; It's just a case of the "interaction" between application, GUI, storage and such that the "persistent display scheme" changes...a software rather than hardware technique...which, again, is why it could really have been adopted from the beginning...the problem, I think, was simply RAM and possibly a simple "lack of imagination" (or, at least, a case of the GUI authors themselves being to "immersed" in the video problems that they failed to "abstract" those considerations away from the application...it's an odd kind of "high-level interface" when the minutia of "repainting" is exposed)...

Though, as Wannabee did criticise, I'd just like to point out that, fundamentally, I'd suggest a "toolmaker's view" for any OS / GUI...that is, at the lowest level, it's very much a "low level" interface, with as "direct" a means to control the graphics hardware as possible...indeed, I'd allow "direct access", basically...then, on top of that, there'd be standardised "drivers" (which the application could directly access)...then, on top of that, this is where the "display list" stuff would be added...

The basic idea being that only "direct access" is, in fact, directly supported in the kernel...then a driver would be a program that makes use of that "direct access" to provide a standardised "interface" to

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

applications (for "portability")...then the higher-level "display list" stuff would be a library that takes the higher-level commands and uses either the "direct access" or the driver, making use of the lower-level interface to do its stuff...

The underlying concept is that, at the lowest level, the OS provides as low-level an interface as possible...then everything else can be "built on top" of that to provide higher-level access (but I wouldn't have it that the driver or the high-level "display list" library are anything "special"...just ordinary programs that "make use" of what's underneath them...the only thing "special" being that they are "standard" with the GUI, so they are always available for use...and, oh, the "driver" program somehow "registers" itself with the OS as performing that duty of being a "video driver", just so that applications can get access to it by asking the OS "please hand us a handle to the video driver, please" :)...)

So, yes, the "display list" is a touch "high-level" but it's supposed be...this is the "high-level" interface for the GUI...if you like, imagine that GDI loaded up Direct3D and then used that to perform its video graphics...that would be the kind of "scheme": The "lowest-level" is literally programming the card directly...then "drivers" would use that "direct access" to actually control the graphics card but then present a standard "driver" interface to other applications and the OS...then, again, the "high-level" graphics library would load in the driver (or possibly also go "direct access"...all levels would be "available" to any program to use, as it desired) and then use that, while providing a "high-level interface" to applications...

Then, as a programmer, you choose the "level" that suits...the higher-level library would be slower performing but "easy" (perfectly fine for word-processors and spreadsheets, as this is "GDI level")...the driver would be less "easy" but faster (games that need "portability" come in here, as this is kind of "DirectX" level :)...and "direct access" is included too (fastest as it's just the application programming the hardware directly but this will be inherently "non-portable"...probably not generally useful for applications because who'd want a game that could only run on one or two select video cards? The real reason that this exists: "Drivers" would also be programs and they'd use this level, obviously...plus, some "control panel applet" that controls a card's "gamma" or "orientation" or other "hardware-specific" gimmicks can use this too...finally, you might just be writing a program for yourself – not "world-wide release" – and, thus, don't care for "portability", just caring solely for your own graphics card :)...if "security" is desired then the OS kernel can stipulate "permissions" for the "direct access" (Linux-esque, in requiring "root permissions" to open up the access...though, to me, even Linux isn't "fine-grained" enough...handing out "root" for just video stuff seems "overkill" (though, this is what X does because it uses "direct access" to the hardware)...so perhaps permissions could be worked in a different way for some "new OS" that was doing things like this)...

I like this "style" of architecture because it caters for all needs,

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

without compromising anyone...and that's what an OS should really strive for...

>>> I've used Flash myself a bit. Using the "tweening" stuff in a GUI is
>>> really a nice idea... And I reserve the right to maybe, possibly,
>>> use it at some time in the future, if you don't mind. :-)
>>
>> Of course not; If I wanted to keep it "secret" then I wouldn't go
>> "blabbing" it around here, would I? I've always said that any ideas I
>> state on the newsgroup, I consider "public domain"...the "reward" for
>> me would be seeing GUIs and such that work this way (or tools that are
>> "incremental" or whatever :) because I do believe it's much better –
>> which is why I talk about these things and explain that – so I'd
>> benefit using your "future GUI-to-be" when you may get around to
>> releasing it...I don't even care about "credit"...pretend it's your
>> own idea, if you like...you know, I consider the `_idea_` its own
>> reward...if it were to lead to lots of great GUIs with super
>> programming tools and so forth then that is my reward: Finally,
>> "freedom" from all the crap that annoys me so much today ;)...
>
> I've been testing around with a GUI system for some time now... One
> problem is, in Linux and Windows, you don't have a good way to access
> gfx hardware without the system's GUI system already running. I. e. you
> need the Windows GUI to be able to use DirectDraw etc., or X (in linux)
> to use GL. Any why would you use a graphical system when you need to
> start X to use it...? Other possibilities are SVGAlib (no hardware
> acceleration) or the linux framebuffer (doesn't work on most hardware).

Yeah; One idea that these OSes don't follow – like the "display list", a different approach that could be used – is that the "windowing" could be part of the driver itself...only the most, most basic kind of windowing: A program "reserves" a section of the screen for output...but the driver does keep track of the "Z order" of these reserved regions...

This would represent the most basic functionality needed for a GUI..."top-level windows" only (but, then again, "child windows" are an organisational convenience...an application "owning" the main window area is already "owning" all the "child window" areas inside that...you don't strictly need any OS / GUI intervention...it would be a "convenience", not a requirement)...and then rather than some "window manager" forcing a titlebar onto every window, the application asks it to add on that stuff as "child windows"...

Okay, so I'm "re-arranging" this around a little...to what purpose? Well, to basically make the GUI `_integral_` to the OS (non-GUI OSes need not apply here)...and that applications can "talk" directly to the drivers...which, coupled with a "peer to peer" architecture (where applications can directly gain access to drivers...well, any "component" can talk to any other one)...we can get rid of the "layered" architecture...that an OS / GUI "wedges" itself into all operations...instead, they "provide services" when asked...

In short, I find GUI OSES like Windows are far, far too "control freak"...this stuff ("portability" and the like) can all be provided in a different way that doesn't require "Herr Fuehrer" Windows wedged in the middle of everything...of course, Microsoft aren't going to do this themselves because they have "commercial interests" (just like we'd never likely see that "UNIX subsystem" on NT because that would be "supporting the enemy" for them...BUT we very well _might_ see this on ReactOS "cloning" NT because there are no "commercial interests" so, you know, ReactOS _welcomes_ all these other "subsystems")...

Hence, you have to consider how much the "architecture" of Windows is "political"...if applications communicated with device drivers directly, in the main, then the hardware manufacturers – who write most of the drivers – could agree "standards" amongst themselves and applications could work to them...and – BANG! – Microsoft loses its "grip" over the situation to "control" it all...hence, even for "DirectX", it's a layer of indirection wedged in the middle...

And, simply, this insistence that the OS is involved in everything is a "political move" in order to ensure that their role remains central...

Consider, say, the BIOS and the VESA routines...VESA are an organisation formed from video manufacturers getting together to form a "standard" because they all benefit from hardware following "standards"...they agree their "standard" and then find an interrupt vector to place it upon...then they publish their "specifications"...applications and device manufacturers follow the "standards" (for their own benefit)...and the point is: Where do Microsoft – even though these may be being accessed from DOS – come into this? Exactly...they don't...

But when you have to call an OS "layer" all the time with API, to access device drivers _through_ Microsoft's API, they then entirely _control_ the game completely...applications have to "conform" to the Microsoft API...drivers have to "conform" on the other side...the API can sit there as a kind of "filter" and "thoughtpolice" on any interaction between hardware manufacturers agreeing "standards" – they can't: It must "conform" to Microsoft's specifications – and application writers, who can only access what Microsoft "deem" acceptable for them to access...

It's a "political" / "commercial" move...designed to "control" what goes on in the entire computer industry...this is why – as well as the benefits – I say "layered" should be replaced by something more "peer to peer"...where any "component" can "talk" to any other one in the system – the OS establishes these "connections" but that's all – not simply for the "performance" (less "layers" to travel through should mean little "overhead") but in order that the hardware manufacturers are put back in control...that it's more "open" that things can perfectly well happen without "Herr Fuehrer" Microsoft or whoever dictating it all...

>> Yeah, if you've used Flash then you exactly know what I mean...because

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

>> this would be kind of using something Flash-like as the GUI's
>> graphical subsystem...and, yeah, you've got it: The "tweening"
>> possibility would really make a large "jump" in GUI appearances...with
>> icons "opening up" into windows and menus animating
>> smoothly...clicking a button and then it "moves up" to become the
>> "title" for the window...
>
> And stuff like that. The MacOS X GUI more or less works that way. e. g.
> if you un-minimize a window from the "dock", it somehow "morphs" to its
> full size. It looks like they're rendering the window contents to a
> texture and then render it on "morphing" polygons, I think.

Yeah, they had the expanding rubber band boxes to show clicks and to "animate" from icons into windows...the rounded corners for windows...all this stuff has been a standard feature of Apple's GUIs...only recently have Microsoft tried to make theirs look a little more "friendly" but it's a poor imitation...

It certainly makes sense these days to probably deal with such things by making use of the 3D hardware, anyway...which is another point about the "display lists" that, you know, add a "third dimension" and, as the GUI itself deals with the rendering, it can choose to use all the 2D and 3D accelerations, as needed, for this kind of thing...

That, perhaps, the GUIs ought to stop looking at it as "2D" and "3D" but use the 3D stuff throughout...and "2D" is just putting textures onto flat front-facing polygons...and the "display list" would already be "appropriate" for that kind of thing...the "paint message" – which really not sufficiently "abstract" from the "mechanics" of the painting – thing is going to be very difficult to switch to a "new model" that really takes full advantage of the hardware that's available...

It's the "myth" with the "backwards compatibility" thing...the reality of the hardware changes that software that isn't willing to also change its view goes "out of sync" with reality...it's a "myth" and "false expectation" that, somehow, you could write this "perfect" program that's so perfectly "abstract" and "compatible" that it'll still be an excellent piece of software in 20 years' time, when hardware is radically different...

It's just crap...for instance, all the 3GLs – C, Pascal, BASIC, etc. – are "sequential" in nature...but there's an increasing move towards "parallelism" (MMX, SSE, multiple pipelines, out of order execution, hyperthreading, etc.) and they just don't naturally fit that model at all...

The bottom line is simply that reality isn't "portable" nor is it "compatible" nor is it "unchanging"...and you can either live in a "fantasy world" that's detached from this reality. messing around with "portable source code" that is DOOMED TO FAILURE because of its denial of this reality that "Utopia programming" does not exist...or you can accept –

Re: Linux, X, ld, gcc, linking, shared libraries and stuff

however annoying it is – that this is simply how it is and work `_WITH_` it, rather than against it...

Beth :)

• *Follow-Ups:*

- ◆ *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
◇ *From:* NoDot

• *References:*

- ◆ *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
◇ *From:* Beth
- ◆ *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
◇ *From:* Johannes Kroll
- ◆ *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
◇ *From:* Beth

- Prev by Date: *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
- Next by Date: *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
- Previous by thread: *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
- Next by thread: *Re: Linux, X, ld, gcc, linking, shared libraries and stuff*
- Index(es):
 - ◆ *Date*
 - ◆ *Thread*