

Re: DirectX in HLA

Source: <http://coding.derkeiler.com/Archive/Assembler/comp.lang.asm.x86/2004-05/0335.html>

From: Bryan Parkoff (*bryan.nospam.parkoff_at_nospam.com*)

Date: 05/17/04

Date: Mon, 17 May 2004 21:23:40 +0000 (UTC)

Beth,

Wow!! You provided a TON of information in this newsgroups. It helps me to understand. I guess that you have a great knowledge of DirectX writer. I suspect that you sound like Microsoft programmer to write and develop DirectX library with DirectX programmers in this team. Didn't you?

--

Bryan Parkoff

"Beth" <BethStone21@hotmail.NOSPICEDHAM.com> wrote in message news:vD4qc.120\$uu2.38@newsfe2-gui.server.ntli.net...

> Bryan Parkoff wrote:

> > It is interesting. It would be much easier to write simple DirectX3D in

> > assembler language like using MASM.

>

> Well, the COM calling mechanisms I believe are simplified for understanding by looking at them in assembly language...

>

> This doesn't automatically mean using MASM is necessarily simpler because, of course, there's more to, say, a 3D space shoot-'em-up game than only calling DirectX (e.g. the rest of the code spinning matrices, calculating shadows, moving all the sprites, synchronising the soundtrack with the action on screen, etc., etc. ;)...and, with a C++ compiler, though HOW it interfaces is made much more complex to understand, you are supplied a set of header files that already deal with the issues and define the necessary constructs that you don't really need to understand it but can just use the constructs in the header file...

>

> It's a HLL: Understanding what's happening often isn't a requirement (what code is inside "printf"? Yeah, exactly...they don't care to tell you, you're just supposed to use it and, well, keep quiet about things after that ;)...the supplied DirectX header files supply the "structures" and C++ actually automatically knows how to make COM calls because it's the same rough way C++ makes its own OOP calls (for C, Microsoft supply a whole bunch of "convenience macros" that hide it away too :)...)

>

> Trying to understand the specifics of the COM "binary standard" calling mechanism is made more complicated because you kind of have to not only understand what COM is doing but also what your C++ compiler is doing too...for example, like the "virtual methods" jargon had confused the poster but what it

comp.lang.asm.x86: Re: DirectX in HLA

> actually represents, really, is a means to "undo" the OOP so
> that the structure is an ordinary uninitialised structure...that
> the compiler won't create a VMT with all the addresses filled
> out because, well, it can't actually do that, as the object is
> _external_ to the program in the DirectX DLLs...in fact, you
> _don't_ use "new" and "delete" to create the objects either
> because the C++ mechanisms are more or less designed for
> _internal_ objects where it has access to all the code inside
> the program and controls the entire process...DirectX isn't like
> that because the "objects" are actually implemented inside
> external DLL files and are "shared" between the program and
> DirectX...this is, in fact, also why all the COM "objects" have
> a "reference count" and a "Release" method because it's the
> _object_ itself which has to become responsible for its own
> deallocation...the ordinary C++ mechanisms of things like an
> implied "delete" for objects created on the stack and so forth
> would not work and would play havoc with these COM objects...
>
> So, in fact, as I say, most of the COM stuff actually relates to
> _switching off_ C++'s OOP, to an extent...you make them "virtual
> methods" so that it's just an ordinary structure and C++ keeps
> out of things with creating VMTs and so forth (one still employs
> OOP, though, basically so that C++ automatically generates the
> "indirection" needed...the C++ OOP mechanisms know to
> automatically push "this" for any "virtual method" held inside a
> structure...so it's used this way just to get that "convenience"
> of having "this" pushed automatically without having to do it
> manually :)...you don't use "new" and "delete" because their
> semantics are designed for _internal_ objects and would likely
> mess up how COM objects would work...blah-blah-blah...
>
> Hence, as I was saying, the irony here is that the C++
> descriptions are highly complicated _because_, in fact, you have
> to play a lot of "tricks" to "undo" much of what C++ does
> automatically because it's no longer useful or applicable to COM
> objects...you put "virtual" in front and "= 0;" at the end of
> every method in the structure to tell it to _butt out_, in fact,
> and NOT create VMTs or anything like that (as they are "pure
> virtual methods" = "pointer to a particular function
> (prototyped) but no actual address yet given to where this
> function is" :)...to treat it as an ordinary uninitialised
> structure that just happens to be full of OOP ("this" parameter
> implied) methods...
>
> > Is it true that OOP is much easier than non-OOP?
>
> They are _different_...it's not a question of "easier / harder"
> but a question of "applicable / non-applicable"...OOP _can_ be
> easier in situations that fit with OOP well but it can also be
> adding on a whole lot of complication to a program for no
> particular benefit that complicates it unnecessarily...
>
> Also, though it sounds cliched these days, OOP is a
> _paradigm_...a way of looking at things...a way of doing
> things...hence, for example, a lot of things which might not
> seem "OOP" initially actually really are in terms of the
> "attitude" and "philosophy"...such as the typical example of
> file I/O in C with "FILE *" handles and passing the "file
> handle" to "fread", "fwrite" and closing it with "fclose" and so
> on...it's a simplistic kind of OOP but essentially it's fitting
> into the "OOP" pattern and the way OOP looks at things and
> implements things...

comp.lang.asm.x86: Re: DirectX in HLA

```
>
> OOP, though, can often bring a lot of extra "fluff" (especially
> when used without really knowing what your compiler is doing
> "under the hood" :)...this is why lots who don't like OOP talk
> about it being "inefficient"...this isn't strictly true as OOP
> can, in the correct circumstances, be very efficient...but,
> yeah, many people using OOP just use it for everything whether
> usual or not and just throw things at the compiler without
> thinking what they'll generate in the final output code...and,
> though, in fact, it's NOT really _OOP_ that's at fault
> here...it's actually the fault of the _misuse_ of OOP...same
> arguments, really, as I make about "portability"...the thing is
> not inherently "right" or "wrong" in itself...it's all to do
> with _USING IT CORRECTLY_ in the right places...
>
> Adding OOP for a program that doesn't really need it (that, in
> fact, is only bloated by OOP and you gain _NO_ benefit for using
> it at all)? Adding "portability" to a program that never uses
> it? Well, when you _MISUSE_ things like this, then, yeah, all
> they are good for is complicating things, slowing it all down
> (both development and run-time performance), bloating,
> mismanaging resources, etc., etc....
>
> Another way to look at it is that OOP (and "portability" and
> other similar devices :) are _TOOLS_...and it would be silly,
> indeed, to say: "Right, I will _ALWAYS_ use a hammer for every
> single job that I do regardless"...yup, a nail in the wall?
> Hammer! (good)...need to strip paint? Hammer! (very bad)...need
> to sand down some wood? Hammer! (terrible choice)...need to
> demolish some big piece of wooden furniture? Hammer! (well, not
> the best tool but it'll do the job...just smash it to pieces!
> ;)...
>
> Kind of getting the basic point? A hammer is only appropriate
> for certain jobs...when used for those jobs, a hammer is "good"
> and "easy"...when used for jobs that it's no use for (like
> sanding down or precisely cutting wood, for example), then the
> hammer is "bad" (if not "absolutely terrible choice" ;) and
> "hard" (if not "impossible" ;)...imagine trying to put up
> wallpaper or paint a wall using just a hammer!! ;)...
>
> OOP groups up related data and procedures into separate,
> encapsulated "groups"...lots of programs are automatically
> naturally formatted in that way and OOP is the right hammer for
> this kind of job...when any kind of program gets "big and
> complex" with lots of different parts to it in all directions,
> then OOP is often a useful way to "order and organise"
> things...put your program into "modules" and then divide things
> up into "groups" and so forth...keep your program code all nice
> and tidy, a bit like putting books on a bookshelf into
> alphabetical order to also keep it all nice and tidy that when
> you want to find a book later, it's very easy to find...you
> know, the thing mothers always fail in teaching their sons:
> "Keep your room tidy!"...when you just pile everything in a big
> heap in the middle of the floor, finding things again is made
> more complex than if you tidied up and put everything in its
> place...see? It wasn't only about appearances, it serves a
> _practical_ purpose to keep your room tidy too! ;)...
>
> > After C/C++ source code is compiled, OOP is ALWAYS converted
> > back
> > to non-OOP into assembler language.
```

comp.lang.asm.x86: Re: DirectX in HLA

```
>
> Well, yes, all code from any language ultimately is compiled to
> _machine code_ ("zeroes and ones", so to speak ;))...
>
> But there seems a little confusion here...if I use structures,
> pointers, tables of pointers to procedures, etc. under C which
> is NOT an "OOPL" (object-orientated programming language), then
> it still IS OOP (object-orientated programming)...as I was
> saying, the C file I/O stuff with "FILE *" and "fwrite",
> "fread", "fclose" really IS OOP...but, yes, C ISN'T an
> "OOPL"...
>
> What C++ does is basically add on some extra keywords here and
> there which automate doing things in an "OOP" way...but, you'll
> note that C++ is otherwise more or less exactly the same as C
> but for these additional "keywords"...C++ is, therefore, an
> "OOPL" (object-orientated programming language...although, to be
> strict, C++ is "multi-paradigm" because it can be used in
> either way...many programmers - me included from time to
> time - often simply use C++ as just "enhanced C" and don't
> bother with the OOP stuff...which is totally okay by the guy who
> invented C++ because he always says that C++ being
> "multi-paradigm" - that you can do OOP and non-OOP, as you
> feel like - is actually its strongest point, in his
> opinion...but we'll call it an "OOPL" because it has OOP support
> built-in and can be used to do OOP all the way, if the
> programmer uses it that way...but we should make the distinction
> as some languages really are ONLY OOP and there's no choice at
> all but to use it everywhere...C++ is more "OOP-enabled C", so
> to speak...as the name makes as a pun: "C plus one"...no,
> clearly Microsoft never understood the joke at all because what
> the hell does "C#" mean? "C with an empty preprocessor
> directive"?!?! Microsoft are so clueless sometimes ;))...
>
> So, OOP code is not really compiled to non-OOP code...not in
> that sense...it's still OOP code in the machine code because
> "OOP" is defined as a way of doing things, not as some special
> "OOP" instructions in the CPU's instruction set or whatever...
>
> You have to make a distinction between "OOP" (object-orientated
> programming) and "OOPL" (object-orientated programming
> language)...OOP just means, as the name actually states,
> programming in a way "orientated" to using "objects"...you are
> formatting out your code into "objects"...that's all "OOP" is
> and you CAN do OOP in ANY language, whether it is an "OOPL"
> or not...and what an "OOPL" - like C++ or SmallTalk or
> whatever - are, are just programming languages where the
> programming style of "orientating" things around creating
> "objects" is BUILT INTO how that programming language works...
>
> Unfortunately, yes, there was an awful lot of "it's something
> brand new!!!" hype going around to sell C++ compilers...and then
> there's some programmers who deliberately or accidentally confuse
> the two things up (deliberately when they are trying to insult
> it: "OOP is crap and bloated and useless"...then, often, they,
> of course, go and put all their variables in a structure and
> then use procedures to manipulate it that takes a pointer to
> that structure and so on and so forth...they are, in fact,
> programming OOP...whether they realise it or not! ;))...
>
> But OOP isn't some kind of "magic"...there aren't any "special
> instructions" in the CPU to do OOP...it really is just an
```

comp.lang.asm.x86: Re: DirectX in HLA

```
> "extension" of "programming practice" ideas like creating
> procedures, systematic naming, grouping related things together
> in a "module", etc...OOP is those ideas pushed a little further
> again...despite what some people say about OOP being
> "revolutionary", it really is NOT...it is clearly
> _evolutionary...an "object" is actually a "module",
> really...being able to merge structures as well as simply
> aggregate them together was a clear _evolutionary_ thing to add
> support for (but call it "inheritance" and suddenly, thanks to
> all the hype, people think it's "magic!" ;)...
>
> It's not really "converted" from OOP to non-OOP...both OOP and
> so-called "non-OOP" are both completely expressible in machine
> code...the instructions aren't different...it's the way they are
> _USED_ that makes it "OOP"...
>
> Perhaps I was misleading earlier...when I said "non-OOP", I
> simply meant "not using OOP jargon or OOP keywords" rather than
> that uses structures and calling through pointers directly isn't
> OOP...what I'm explaining _IS_ OOP...but what I meant with
> "non-OOP" was "explaining it _WITHOUT_ using OOP jargon or
> special OOP keywords"...the code itself _IS_ OOP because that is
> defined _solely_ by _what it does_ (and, yes, the dividing line
> _isn't_ always clear and can be "fuzzy", exactly because it's
> about how a program works rather than what instructions or tools
> a program uses to do it...but you can get an appreciation with
> enough exposure to be able to look at something and say "yeah,
> that's definitely OOP!" ;)...
>
> > Do you think that there is a possibility that DirectX3D is
> written in
> > C/C++ source code may only improve 70% performance while
> DirectX3D is
> > written in assembler source code may only improve 90%
> performance? Which is
> > faster? The answer would be DirectX3D that is written in
> assembler
> > language.
>
> Right, it's not totally clear what you mean here, sorry...
>
> But if you mean using C/C++ or ASM source code to _access_
> Direct3D from your program then, actually, there's likely to be
> _NO DIFFERENCE WHATSOEVER...what you'd be doing in the ASM is
> only what the C++ automates for you (C is NOT an OOPL so, in
> fact, would have to do things "manually", exactly like ASM does
> and we shouldn't really say "C / C++" in this context anymore
> because this is where C and C++ go their separate ways ;)...you
> can't get it any smaller than pushing the parameters, pushing
> "this" and then making an indirect "CALL" instruction via the
> "table of pointers" (the "VMT" in the OOP JargonSpeak
> ;)...that's what you'd code in ASM and that's what a C++ would
> almost certainly compile it down too...
>
> There is, actually, _NO REAL ADVANTAGE_, generally, to using ASM
> than C when it comes to _making calls to libraries_ (e.g. Win32
> API, DirectX, etc. ;)...this isn't where you make any real gains
> over HLLs...the reason basically being: Win32 and DirectX and
> OpenGL and all these other libraries use _HLL conventions...and
> when you're using ASM to access them then you actually _must_
> follow the "HLL conventions" for calling into the
> libraries...and when you follow the "HLL conventions", you are,
```

comp.lang.asm.x86: Re: DirectX in HLA

```
> of course, simply copying what the HLL does exactly...because
> of this, ASM has no great advantage over HLLs for calling into
> libraries with "HLL conventions" on their procedures...these
> libraries, really, are HLL libraries...it's just because ASM
> is capable of interfacing to anything that you can "fake up"
> or "emulate" or "copy" the same calling sequence manually that
> the HLL compiler would do automatically so that the HLL library
> is none the wiser that it's ASM code rather than HLL code
> calling it...
>
> ASM loses its advantage when calling into libraries with HLL
> calling conventions because it has to temporarily "stop being
> ASM", so to speak, and start acting like a HLL - "pretending" to
> be HLL code by copying exactly what the HLL compiler would
> produce - to work with the library, before it can return to
> being ASM once more...well, in a manner of speaking, anyway...
>
> There is one place where ASM can win over HLLs in calling
> sequences but it's not all that common to be of great
> advantage...for example (pay attention, also, those people who
> use "invoke" all the time and think there's no difference or
> advantage to not using "invoke" than to using it...you might be
> "missing a trick" here ;)...
>
> Rather than (as a HLL compiler or someone using "invoke" would
> tend to produce)...
>
> -----
>
> .code
>
>     invoke GetModuleHandleA, NULL
>     mov     edx, eax
>
>     invoke GetCommandLineA
>
>     invoke WinMain, edx, NULL, eax, SW_SHOWDEFAULT
>
>     invoke ExitProcess, eax
>
>     end _start
>
> -----
>
> (...which would expand into...)
>
> -----
>
> .code
>
>     push    NULL
>     call   GetModuleHandleA
>     mov     edx, eax
>
>     call   GetCommandLineA
>
>     push   SW_SHOWDEFAULT
>     push   eax
>     push   NULL
>     push   edx
>     call   WinMain
>
```

comp.lang.asm.x86: Re: DirectX in HLA

```
>     push    eax
>     call   ExitProcess
>
>     end _start
>
> -----
>
> ...ASM can do one thing that HLLs tend not to do (and that
> "invoke" gets in the way of ;) of "interweaving" API calls where
> the return of one API call feeds directly as a parameter in
> another call...like the following little change:
>
> -----
>
>     .code
>
> _start: push SW_SHOWDEFAULT
>
>     call GetCommandLineA
>     push eax
>
>     push NULL
>
>     push NULL
>     call GetModuleHandleA
>     push eax
>
>     call WinMain
>
>     push eax
>     call ExitProcess
>
>     end _start
>
> -----
>
> Which is slightly - not a great deal - better...smaller, doesn't
> use any registers (so those could be used to store things for
> other purposes...registers are precious to look after ;) and
> some slight improvements here and there...
>
> Though, unlike the above, one has to be more careful to ensure
> the stack is balanced, which is trickier when the stacks for the
> parameters are all being "interweaved" like this...
>
> BUT, generally speaking, when calling _HLL libraries_ (which
> things like DirectX are because they use the HLL "STDCALL"
> conventions and stuff :), ASM gains you little advantage over
> HLLs like C / C++...ASM makes all its gains _elsewhere_ but,
> unfortunately, can't do anything about Microsoft's crap HLL
> designs, that insist you have to do things in a "HLL way",
> whether using a HLL or not, whether you care about "portability"
> or not...indeed, this is why I moan elsewhere that an OS
> shouldn't really do that...it should present an _ASM_ interface
> to things and then C++ compilers can have "wrappers" to make
> things "HLL friendly" (this would, in fact, be a _good_ solution
> for HLLs too because rather than saying "STDCALL" for all HLLs,
> be they C or Pascal or whatever, then the "wrappers" can be
> custom-built and tailored to the language being used...C
> "wrappers" using C conventions, Pascal "wrappers" using Pascal
> wrappers...it wouldn't just be good for the ASM people but
> better for the HLL people too...this is why I say this method
```

comp.lang.asm.x86: Re: DirectX in HLA

> makes the most sense for an OS _whatever_ language you're
> using...an OS should always be designed in that way, in my
> opinion ;)...and, thus, coding ASM? Don't need to put up with
> "portability" nonsense _that you don't use...note, even many C
> or Pascal coders or something could do the same too - to improve
> their performance too - by creating some "inline ASM macros"
> that by-pass the need for "wrappers" ("inline ASM" is, indeed,
> "compiler specific" but most C / Pascal compilers and stuff can
> actually do it...losing "portability" is okay because if you're
> by-passing the "wrappers" to go straight to the registers then
> that's "non-portable", anyway...every CPU has different
> registers, after all...that's the point here, you see: "those
> who want portability can get it, no problem - use the wrappers -
> but those who would benefit from dispensing with 'portability'
> to get better performance - games, multimedia, etc. - can
> 'by-pass' it at their discretion" ;)...and if a HLL uses one of
> these "wrappers" then its performance only really drops to _what
> it is already now...in a sense, _everyone_ is forced, under
> Windows, to use some "generic wrappers"...well, I would say that
> they should be separated out so that the programmer _CHOOSES_
> whether they want the "wrapper" or not...but, anyway, all
> pointless talk because Microsoft do it the wrong way and will
> almost certainly never change from that now they've made their
> choice to be "HLL" all the way about everything...
>
> And if you meant that DirectX _itself_ would be improved by
> writing it in ASM rather than C / C++ then, yeah, probably it
> could be speeded up...but I'd wager, in fact, _NOT BY THAT
> MUCH...because DirectX actually just sits on the device drivers
> as a kind of "filter" for the device drivers..._ALL THE HARD
> WORK_ of DirectX happens inside the video display driver, inside
> the soundcard device driver, inside the mouse and keyboard
> device drivers, etc....
>
> In truth, "DirectX" is a "correction" of a basic design flaw in
> the original Windows...of course, this doesn't very clever or
> good that you'll never hear Microsoft phrase it that
> way...basically, the original Windows (not actually too
> dissimilar to other GUIs...most GUIs, in my opinion, are
> _wrongly structured_ because they copied Xerox's original
> design...and what's Xerox's strong point? _Specific embedded
> designs_ tend to be what they mostly do inside their
> photocopiers and stuff...specific purposes devices their
> speciality...and I would say that those writing _general
> purpose_ GUIs for _general purpose_ PCs perhaps should NOT have
> taken Xerox's designs so _literally...but rather as a
> "guide"...but what is done is done, eh? ;)...the original
> Windows could only access the display through the built-in
> GDI...and GDI was NOT designed to be "real-time" drawing tons of
> textured polygons...in fact, GDI is NOT even really
> well-designed for simple "pixel plotting"...GDI was designed as
> a "friendly library" for, well, programmers who "aren't very
> good with graphics" to write spreadsheet programs...GDI, for
> example, works just as well with printers as it does with video
> screens and in being "generic" for both types of quite different
> design, it had to _compromise_ certain useful video
> functionality...yup, GDI makes "WYSIWYG" ("what you see is what
> you get" ;) functionality that what is on the screen prints out
> to the printer exactly the same, totally easy and
> "built-in"...so, great, makes spreadsheets and wordprocessors
> easier to write, then...but this design _compromises_ out the
> possibility of flicker-free 60fps textured polygon computer

comp.lang.asm.x86: Re: DirectX in HLA

> games...they just didn't think of it and didn't work it into the
> design...
>
> In fact, what games programmers (and demo programmers and those
> with need for high-performance multi-media of any kind ;)
> actually wanted, really, was to just be able to load up the
> device drivers and talk to them directly...some kind of, for
> example, "GetVideoDriver" API which provides a "handle" to the
> driver handling the video display and then you could use some
> "SendMessage" API to send commands directly to the driver to do
> things like change video modes or load a texture into video
> memory or draw a polygon or whatever...of course, same idea with
> the soundcard...a "GetAudioDriver" API and then you communicate
> with the driver directly...
>
> Note that this is a perfectly valid possibility because device
> drivers are already "portable" in the interface that they show
> to the OS (that's how the OS themselves can use the hardware in
> a "portable" way ;)...and, under this design, GDI would,
> instead, simply be a "user code library" that provides a
> "high-level" API like it does now and inside the library, all it
> would be doing is handling all the device driver communication
> on your behalf...to be a "wrapper" for talking to the device
> drivers directly with a "friendly" interface of "DrawBox",
> "DrawLine", "ChangePen" and that kind of GDI vector graphics
> stuff...indeed, easy enough too, that this "wrapper" could be
> clever enough that it could provide the same "API" to both a
> video display and a printer...so, indeed, I'm NOT talking about
> getting rid of GDI at all...but it should have been strictly a
> library on top of the device drivers...
>
> Unfortunately - and Windows is far from the only GUI OS to, in
> my opinion, make this "mistake" - Windows was designed with GDI
> at the front...the device driver interface was designed without
> any idea of making it "friendly" for application programs to
> use...no, instead, device drivers were only for the OS to deal
> with...applications must talk to the OS who passes it onto the
> device drivers...
>
> BUT, of course, game designers didn't like this at all...and, in
> fact, games STILL kept being made for DOS, well after Windows
> had become the main OS on PCs for people to do word-processing,
> spreadsheets, play Solitaire (the only undemanding game that
> could happily live with using GDI only ;) and that kind of
> thing...Microsoft realised that they had to do something to
> fix this problem...
>
> Hence, "DirectX" was stuck onto Windows as an "extension" where
> you pass it more "direct" and "specific" commands that it will
> "by-pass" the usual API nonsense and take more directly to the
> device drivers...it's just a "filter" to get around a bad design
> in the first place...
>
> And anyone who disputes that the original design was not a bad
> design (IF there is anyone)? Proof of the pudding is in the
> eating...they HAD TO create DirectX to cater for this or
> Windows would have remained "shunned" as a gaming and
> multi-media platform...
>
> And, now, in hindsight, anyone creating their own OS from
> scratch should consider learning from MS's mistakes rather than
> repeat them themselves...put your "DirectX" equivalent at the

comp.lang.asm.x86: Re: DirectX in HLA

```
> _core_ of the OS...in fact, design it so that it's a reasonable
> thing for an application to get a "handle" to a device driver
> and talk to the device driver directly (still need device
> drivers more generally because of the need to _regulate_ in a
> multi-tasking environment - can't have everyone attack the
> screen at the same time! - and to also provide "portability", as
> PC video and sound cards and stuff are all completely different
> that it's a _practical_ problem that you need something like
> this...unless you use VGA mode 13h - not good enough a standard
> for today, unfortunately - for everything, anyway ;)...you can
> provide "GDI" and "OpenGL" and all that kind of thing as "user
> code wrapper libraries"...they don't have to be lost, you see,
> just that they should be the "extensions" rather than the core
> of the OS with using the device drivers more directly as the
> "extension"...it's simple logic: build it up bit by bit, one
> component using the lower-level component...don't just jump from
> "sending command on I/O ports" to GDI in one big leap...it's NOT
> as clever an idea as it might first appear to do that...the
> other OS have learnt this big-time having to "replace" their
> main, crappy graphical interface with better "extensions" bolted
> onto the side...
>
> > The fact is that C style rather than C++ style is
> identical to assembler
> > language however C++ must be converted back to C before it is
> compiled into
> > assembler language. Please correct me if I am wrong.
>
> Ummm, not completely sure what you're saying that I can't
> correct you or otherwise, until I'm sure I know exactly what
> you're trying to say...
>
> Right, "C style [...] is identical to assembler language"...yes,
> C has no "OOP" constructs built into, so the approach with C
> (not C++) is actually very, very much like ASM...that is, you
> just create a "structure" and then _manually_ pass along the
> "this" as the first parameter and all the same things as in
> ASM...
>
> C++ is just different because it recognises and understands what
> "methods" are and, thus, knows to automatically push "this" as
> the first parameter (it does this "behind the scenes" :)...so
> you don't need to put "this" into the parameter list and when
> you use something like "pDirect3D -> CreateDevice()", then C++
> automatically knows that "this" (pDirect3D itself, in fact ;)
> should be automatically pushed as the "hidden" first
> parameter...if you like, the C++ is "OOP-aware C"...it realises
> that this is an "OOP thing" and knows that OOP needs the "this"
> pointer as one of the parameters...so it does this all
> automatically so that it's "hidden" and you don't need to know
> about it...
>
> Although, in this case, that's _all_ C++ does that's
> advantageous for the programmer...and, to get it to work right
> with _external_ COM objects, you wonder if this is worth the
> price when you have to put "virtual" in front of everything and
> "= 0;" after it and then list the "inheritance" at the top and
> so on and so forth...in this case - for COM objects - C++
> probably requires more "tweaking" to work properly with COM than
> you actually gain from having "this" hidden from you (that's all
> you're doing all that stuff for...to have C++ do the "this"
> stuff automatically...and, I'd say - but then I'm an ASM
```

comp.lang.asm.x86: Re: DirectX in HLA

> programmer and "biased" so I probably would say this - that
> there's no great point in "hiding this", anyway...what's so hard
> about having a pointer as the first parameter of a procedure and
> remembering to push it? It's not like it's a difficult value to
> obtain...you're already using it to find the "method table" to
> make the OOP call, anyway...that is, in "pDirect3D ->
> CreateDevice()", you've already got your "this"...it's the
> "pDirect3D" value itself! You go through all this effort,
> really, to stop yourself having to type out "pDirect3D"
> twice...and that's about it in reality...so, as I say, it might
> actually be easier and better not to use an OOPL for this
> kind of thing because it demands lots of "tweaking" to do
> something with "hiding this" that's hardly worth all the
> effort...certainly true for low-level programmers - the MOST
> LIKELY people use DirectX, if you stop and think about it for a
> second because it's used for high performance games and
> multimedia applications (so, if you're not particularly good
> with "low-level" things and don't really understand it too well
> then, ummm, I wouldn't totally say "what are you doing
> programming DirectX?" as it can benefit even VisualBASIC
> programmers - after all, their HLL code ain't the best so they
> need every bit of help on the "speed" front they can get - but
> it called into question what the designers of these things are
> often thinking...you know, "yes, we expect them to understand
> matrix mathematics completely...but, no, they do need 'help' in
> learning how to use their mouse and which button on their PC is
> the 'power button'"...I mean, either these people know what they
> are doing and are able to learn to know what they are doing...or
> you treat them like complete morons...what we actually get,
> though, is some really bizarre "in-between" where you're
> expected to be an expert in highly complex game algorithms but,
> at the same time, you're also considered a "blithering idiot"
> that, ooh, pushing "this" makes your brain explode with
> "Overload! Overload! Pushing a pointer too complicated! Pushing
> a pointer too complicated!"...yet, by the way, you're fully
> expected to understand how to use and manipulate pointers in
> other places without any problems...it's a confused design from
> very confused people at Microsoft, who aren't sure if this is
> for "Clueless Newbies" or "Advanced Coders" or whatever, so
> weirdly try to make it all those things at the same time in
> different places with no consistency at all! Indeed, it's almost
> a catchphrase now but: "Bloody Microsoft!" ;)...

>
> "however C++ must be converted back to C before it is compiled
> into assembler language"?

>
> Ummm, no...as mentioned by Alex, modern C++ compilers compile to
> object code directly...no in-between "C" phase...the very first
> C++ compilers did do it "via C" but that was more just a way to
> get the language working at first until a full C++ compiler was
> written (HLA is really doing the same thing in converting HLA to
> MASM or FASM and then using them to assemble the code...it's
> just a "quick and nasty" way to get a new language up and
> running without writing the full compiler...Randy did it this
> way so that he could spend most of his time on the design of
> the HLA language to get it right - to "prototype" it - before
> actually diving in to code it up properly...but HLA v2.0 will no
> longer need an "intermediate" assembler to help out...it's just
> a way to test out designs with a simple "prototype" until you're
> sure you've got the language properly sorted...then you can
> write it up properly as it's own compiler, safe in the knowledge
> that you've got the language designed exactly the way you wanted

comp.lang.asm.x86: Re: DirectX in HLA

```
> it to be :)...
>
> And assemblers and compilers compile to _machine
> code...assembly language is the "human-readable" form of
> machine code and, in many contexts, it doesn't which word you
> use...but, strictly, in this context, we're compiling to
> _machine code_ (inside "object code" files most usually
> ;)...and, no, most C++ compilers go straight to that machine
> code without converting to C or anything...
>
> Now, if you mean "C _style_" rather than literally C...that is,
> "converting" C++ OOP into C non-OOP before outputting the
> machine code...that's not true either..."OOP" is perfectly
> expressible directly in machine code...it just goes straight
> from OOP to machine code...again, "OOP" is the _STYLE_ that the
> code follows...you can do "OOP" in C just as well as C+...the
> difference with C++ is that it _understands_ OOP automatically
> and has special "OOP keywords" to make OOP programming
> easier...it's an "OOPL"...
>
> > If there is no data struct, it must be done inside
> function to push and
> > pop data in order to balance the stack.
>
> Again, not totally sure what exactly you mean here...what does
> "no data struct" refer to here? Sorry...I think you're getting
> the ideas about the programming stuff right here but your
> English is a little hard to read in places...be patient and
> tolerant with me in trying to understand: I'm too stupid and
> stuck with native English thinking that I'll need some more of
> your help to understand completely...
>
> But, anyway, the "balance the stack" stuff is, of course,
> correct in that pushes and pops must be "balanced" by a
> procedure...
>
> Let me think...do you mean - as C++ allows - for an "object" to
> be put onto the stack rather than in the "heap" memory or data
> section? Is that what you mean by "no data struct"?
>
> If so, then I see what you mean...and what happens is that the
> function doesn't really "push" and "pop" the parameters on the
> other end...instead, it uses the "this" pointer - which will
> have the address of the "struct", whether that is on the stack
> or in the data section or whatever - to access the data...this
> is actually part of the reason _why_ you must the "this" pointer
> as the first parameter to allow the function to be able to
> access the "struct" _wherever_ in memory it is...because by
> passing the pointer to the struct - the "this" pointer - then
> you pass it the _address_ of the struct, whether it's on the
> stack or in a data section or dynamically allocated "heap"
> memory or wherever it is...and it doesn't actually touch the
> stack with "pop" inside the function to allow this to
> happen...what it does is use "esp" - the stack pointer - to
> access the parameters and the "this" pointer - the first
> "pointer to struct" pointer - to access the struct...because
> these are just "addresses" then they could be anywhere in
> memory...
>
> > How is it possible that stack can handle more than two data
> structs.
> > For example, first data struct has an address 0x0012aa00 and
```

comp.lang.asm.x86: Re: DirectX in HLA

```
> second
> > data struct has an address 0x0012bb00. Is it the way that
> first data
> > struct can be pushed into stack before it is used, and then it
> is
> > popped out of stack. Second data struct can be pushed into
> stack to use.
>
> Well, it's able to handle having the "structs" anywhere in
> memory because, of course, we're passing the address of the
> "struct" as the first parameter ("this")...so, wherever the
> "struct" is in memory, the "this" pointer is what the function
> uses to find it...in other words, the function itself doesn't
> "work out" where the structs are...we TELL IT where they are
> with "this"...that is basically WHY we pass "this" as the
> first parameter and why "this" is simply the address of the
> start of the struct...because, yeah, how could the function work
> this kind of thing out? It couldn't really...so, instead, as a
> programmer, we TELL it where the "structs" are by passing the
> address of the structs as part of the function call
> itself...and, yes, if you were to pass the wrong address then
> this scheme wouldn't work and you might blow up the program (in
> truth, that is the reason why C++ tries to "hide" the first
> "this" parameter from the programmer...so that it deals with it
> and makes sure the addresses are always right...to eliminate the
> possibility of the programmer making a terrible "bug" of passing
> the wrong pointer to the function - which actually doesn't know
> where the "structs" are at all and totally trusts the pointer
> you provide as being always correct - and crashing the program
> ;)...
>
> When there are two structs on the stack then, by definition, the
> struct pushed first will be "alive" while the second struct is
> on the stack...because to get back "down" the stack to the
> struct pushed first means popping off all the stuff above it on
> the stack and, hence, the second struct has to be popped off
> first from the stack to get to the first struct (in that
> description, I've said "up" and "above" in the contexts of
> real-world stacks - like stacks of plates - inside the computer,
> though, stacks are actually upside-down...yeah, as if it wasn't
> complicated enough, they go and put our stacks upside-down to
> confuse us! No, there is a good reason for that, actually, but
> it's another post in itself to start to explain how weird stacks
> are ;)...
>
> In practice, though, a program would not normally be putting its
> COM "structs" onto the stack...you CAN do that, of
> course...just make enough room on the stack for the "table of
> pointers" and then pass the stack pointer as the address...and
> then the addresses would be filled into the stack
> space...indeed, C++ completely supports objects on the stack and
> whenever you declare an object as a local variable, then this is
> actually what it does...
>
> But, generally, in OOP, the memory for the "structs" is usually
> dynamically allocated "heap" memory (allocated with
> "VirtualAlloc" or "malloc" :) because part of OOP is to allow a
> programmer to make lots and lots of the same "object" (like,
> say, loads of "texture" objects in Direct3D for drawing textures
> on all of your polygons...one "texture object" for every texture
> you want to draw in the "scene" on the screen :)...and you often
> have no idea in advance how many objects you want to allocate
```

comp.lang.asm.x86: Re: DirectX in HLA

```
> while running the program...it's a "variable-length" thing
> usually...and, thus, allocating "heap" memory tends to be the
> usual way to store the "struct" memory because you can allocate
> as many structs from this memory as you have spare available
> RAM...must remember to deallocate that memory when you're
> finished, though...
>
> The stack - as "local variables" - would be more common but, as
> you seem to be talking about - you have to make sure that your
> "object" doesn't just "vanish" because, oops, it gets popped off
> the stack before you've finished using it (a nasty bug because
> you've probably NOT called "Release" on the object as you're
> supposed to do when you finish and objects are responsible for
> deallocating themselves in COM...in other words, you would need
> to be a little careful not to make a bug here and cause a
> "memory leak" that you have an "object" sitting around in memory
> that you've lost the pointer to - so you can't deallocate it
> anymore when you've lost its address (oops! ;) - and that won't
> disappear until it is told to do so with a "Release" call...it's
> not so bad in that Windows should really clean this stuff up
> when the process ends...but, while your program is running, then
> it could have a nasty "memory leak" in it...yes, when using
> "pointers" and "dynamically allocated memory" then you always
> need to be wary and keep an eye for any "memory leaks" and be
> careful about that stuff :)...
>
> Static memory - memory in your data section - is usually used
> for some objects...for example, you only need to have _one_
> "IDirect3D" object - this is the "object" that just represents
> "Direct3D" itself (hmmm, Microsoft haven't read the OOP
> textbooks about not creating "super objects", have they? Why am
> I not surprised at anything Microsoft do? Sorry, ignore this
> comment, it isn't relevant...I just spotted yet another stupid
> thing Microsoft have done...again! ;) - and, therefore, you'll
> only ever need one "struct" for this "IDirect3D" object...so,
> might as well just create a static struct in your data section
> for that (well, unless you're _really_ fussy_ about "file size"
> ;)...in fact, if your program always uses a fixed number of
> "objects" then you can, indeed, do it all with static memory -
> the ASM DirectX demos I've seen only usually consider doing this
> way and don't use the stack or dynamic "heap" memory or
> anything - from start to finish...but a big, complex DirectX
> program would likely need to do things in a "flexible" way and
> use dynamic memory (or, indeed, the stack...which is actually
> also "dynamic memory" but a "special" kind that works in a
> certain way...and also keeps track of "return addresses" from
> procedures that you've, indeed, always got to be careful to
> "balance" the stack or it'll pop off the wrong address and then
> jump into the middle of nowhere in memory and, almost
> certainly - bang! - crash badly :)...
>
> But, in practice, this usually isn't a problem because the
> program logic means that you normally wouldn't run into problems
> unless you were doing it in a really weird way...you know, a bit
> like how you never have to worry about the stack popping off
> procedure "return addresses" in the right order normally because
> of the way most programs "CALL and RET" in a strict sequence
> that the stack is quite naturally always "balanced"...
>
> And, seeing as we're talking about "DirectX in HLA" (well,
> according to the subject line we are ;)...then it's worth
> looking over the chapter about the stacks...but also the stuff
```

comp.lang.asm.x86: Re: DirectX in HLA

```
> about "displays"...this is a stack mechanism that's not used in
> C / C++ at all, so it might be new to many coders who've only
> done C / C++...HLA (as well as Intel's "ENTER" and "LEAVE"
> instructions before Rene launches into an "anti-assembly"
> rant...sorry, supported in the CPU directly with its own
> instructions so quite "assembly", in fact...just because C / C++
> doesn't do something is automatic that it's not possible or
> useful or "anti-assembly" or whatever ;) can create "displays"
> for procedures so that procedures can actually access the "local
> variables" of other procedures (access the other procedure's
> "stack frame" without pushing or popping or anything :)...those
> chapters contain a lot about "stacks" that's useful to read up
> about...in fact, as noted, C / C++ doesn't support "displays"
> that, actually, HLA and the AoA chapters about this stuff is
> something that many other textbooks don't talk about at all
> because their authors only really know C / C++ and haven't heard
> of "displays" to write about them...
>
> > Is it the way how DirectX3D work?
>
> Well, I'm not 100% sure that I completely understand your
> question, so maybe I've given you the wrong answers here
> (indeed, as you said to me: "correct me if I'm wrong" ;)...but
> it's more the way C++ works _when using Direct3D_ than the way
> Direct3D itself works...Direct3D doesn't really care _where_ you
> have your "structs" because you have to provide a _pointer_ - a
> memory address - to it and Direct3D just uses the memory pointed
> to by that address...it doesn't care nor does it even know
> whether that memory is on the stack, in the data section or
> "heap" memory or whatever...it's one of the reasons _why_ OOP
> works the way it does with "this" pointers being passed as a
> parameter all the time to make sure that it can work with _any_
> kind of memory...
>
> Beth :)
>
```