

Re: Can this loop be made faster ?

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-02/1228.html>

From: Beth (*BethStone21_at_hotmail.NOSPICEDHAM.com*)

Date: 02/28/05

Date: Mon, 28 Feb 2005 03:27:36 GMT

wolfgang wrote:

> *The Half skrev:*

> [...]

> | > *but I'm afraid you have to use the windozeAPI or DirectDraw,*

> | > *not sure if you can use code like above there.*

>

> | *One can use such code, but only in a backbuffer, which would then be*

> | *copied (blitted) by a hardware blitter to the screen. But one would*

> | *probably prefer to use the hardware blitters even to work on the*

> | *backbuffer, as the full operation of locking the the memory, reading it*

> | *from the hardware, changing it, and copy it back to the hardware would*

be

> | *slow. As far as I know you can do this in DirectX. I did this once, and*

it

> | *made otherwise smooth graphics become jerky.*

>

> *Yes.*

Ah, basically, the hardware is now designed to work with the 3D stuff directly...you know, "optimised" for it...

Hence, you can grab the "back buffer" (or any "surface" on which to draw) and then "lock" it, which returns a pointer to the memory for direct writes..."unlock" it and then do your "flip" or "blit" or whatever you need to do to make it visible on the screen...

BUT, in practice, the modern cards are so "rigged" for the polygons and textures (and having to "lock" and "unlock" the "surface" all the time has "issues" because it's implicitly a "serialising" thing, of course...that is, whilst "locked", everything else is waiting on the CPU to finish and "unlock" it again, so that any "parallel" optimisations are screwed up by that...and because the CPU needs `_access_` to it then there's the "issue" about having to pass all the data across the bus to the videocard or, alternatively, using a "system memory" buffer but then needing to "blit" that over to the card via the bus...and so on and so forth :)...that, in fact, the best approach even for the 2D stuff is to create a polygon and a texture for it then write your graphics onto the texture...but just make it face the screen "flat" so there's no "perspective"...a kind of "2D using

alt.lang.asm: Re: Can this loop be made faster ?

3D" thing...

Basically, they are so "optimised" for 3D operations now that even 2D games perform best when you "work with" the 3D hardware but just "fake up" 2D by ignoring "depth" and making all the polygons "face front"...you know, "go with the flow" of the 3D optimisations because things have now proceeded far enough that these videocards are "thinking 3D" throughout that trying "direct 2D" is actually "awkward" for the card and doesn't perform as well...

Indeed, in DirectX, you have to set a "flag" to say that you want to be able to "lock" the "surface" for direct writes...because the "default" is now to assume that, most times, people don't want this anymore...setting the "flag" basically makes sure that the memory buffers are allocated "within reach" of the CPU to allow that...otherwise, there are all these "texture managers" and things which – correctly, for 3D performance, of course – try to get as many of the textures _onto_ the videocard as possible because then it can "blit" them without needing to pass graphics over the bus (the "bottleneck" in all of this)...

Mind you, these days, it's _hard to tell_ what exactly is going on because, of course, DirectX is all "abstract"...so the way DirectX is set up is, of course, "lowest common denominator"...there is mention in the documentation that, for example, "locking" _might_ be a performance killer...the "might" there suggesting this is "depending on what the underlying hardware is and what support it offers in the driver"...for good performance across most cards, though, the "2D using 3D" is now the preferred option...the cards are just now so "rigged" for 3D with "special optimisations" and built-in "texture management" and that kind of thing that using the 3D but "faking" it to do 2D performs the best...

Although, of course, there's lots of "variables"...this is presuming the graphics are mostly "static"...not "dynamically generated"...there is "dynamic texture" support but it goes back to the point that to give the CPU "access" for direct writes means that the buffers have to be placed in system RAM and then "blitted"...and that's the actual "bottleneck" problem here: If you've got some 128MB videocard, then you can load up the card's RAM with lots of "texture" graphics and then the card can "blit" from its own internal RAM very quickly...whenever the CPU is involved, though, it sits on the other side of the "bus" and requires doing it in "system RAM" and then "blitting" things over...you can see the problem, I'm sure...as the bus is constantly involved, it all slows down to "bus speed"...while, if you can "go with the flow" of using 3D "textures" and stuff, you can "pre-load" lots of your graphics onto the videocard itself and the videocard can "blit" its own video-memory-to-video-memory things very quickly with internal "optimisations" for it...

Hence, for performance, the typically "best" way is to load up your "sprites" as textures into the videocard and then the built-in "texture mapping" things in the card itself can render those super-quick onto polygons that you always keep "facing front"...

Re: Can this loop be made faster ?

alt.lang.asm: Re: Can this loop be made faster ?

> / *But its been almost two years since I did any DirectX programming, and*
> / *that was for DirectX7, so much may have changed by now.*
> /
> / *Of course, now theres is pixelshaders, that can do this instead, I*
> / *_suppose_, but I have not tinkered with those yet, allthough I have*
> / *several megas of unread docus about it. Time*
permits....well....(looking
> / *blank into the screen for a moment...feeling the strains of*
insomnia...)
> / *As far as I have understood, the hardware pixel shaders allow for us to*
> / *upload some small pixel shading code to the hardware, and so the code*
will
> / *be called, by the hardware for each pixel that is rendered, after the*
> / *texture has been applied. Its supposed to create possibilities for*
unreal
> / *graphics effects. but I am really blank to the details yet. But from*
> / *looking at the really amazing games. (just played HL2) theres really*
been
> / *an awsome revolution in game graphics. It now looks so wonderful, that*
it
> / *almost ready to take *away* the magic. :)) Oki thats just my sour*
> / *shoulders, and broken ego speaking.... No, actually the new graphics*
and
> / *physics engines are really amazing. They make moch, of just about*
anything
> / *else. Soon we will simply be able to be part of the movie. I see an*
> / *amazing industry growing up, giving us interactive movies, and awsome*
> / *games. New ways to tell stories. We should try to keep track of them,*
but
> / *sooner or later they may decide to look us out.*
>
> / *If just all this new hardware would come with detailed documentation*
> / *instead of HLL-created windoze-drivers...*

Actually, actually...the "pixel shaders" are pretty interesting things...first off: You program them with a special "pixel shader assembly language"...

Basically, what the videocard people have done is that they have _programmable_ GPUs...little dedicated CPUs for the graphics...and the "pixel shaders" have a set of "registers" and operations that you can perform on the "registers"...every time the GPU renders a pixel, then it can run a little "routine" (in this GPU "assembly language" :) to work out what colours and things to make the pixel...

So, in fact, what's effectively happened is that the system has effectively become "multi-processor" and that's a very positive thing, to be honest...if you like, they've "exported" a little mini-CPU onto the videocard itself and the "pixel shaders" are now offering _programmable_ "assembly routines" to be uploaded onto the videocards to "customise" how the CPU renders the pixels...there's also "vertex shaders" which are much

Re: Can this loop be made faster ?

alt.lang.asm: Re: Can this loop be made faster ?

the same thing but with the 3D "vertex" points (you know, you could upload some "routine" that smoothly "morphs" one 3D shape into another...and this would all be happening on the videocard itself so it's all "super-smooth" and runs in parallel with the CPU, freeing it up for other things :)...

But, yes, we unfortunately still have to go "via" DirectX...because it's all "abstract", I don't actually know how "standardised" this "assembly language" is in the actual hardware...if it was fairly "standard" then there would be an argument that DirectX should perhaps "give up" this stuff...you know, if it's "multi-processor" now then treat it that way...write assembly code for the CPU and write assembly code for the GPU...just as if you've got multi-processors (because you actually have: They just aren't "equal", as the GPU is a "graphics specific CPU" :)...but I suppose the problem is the same old problem...there's all these different graphics cards out there and if they are all inherently different in the actual hardware then catering for them all with "one routine for an ATI 8000, one routine for an ATI 9600, etc., etc." this would be impractical and infeasible...

Ah, it's the same old, same old, wolfgang...nVidia, of course, have their "no documentation" policy and "binary only" drivers...and they don't seem to want to give that up...

But the "pixel shaders" are actually, for once, heading back in the correct direction...this isn't the "total solution" I've been talking about before but, basically, it was, indeed, annoying that the cards were all "polygon accelerators" and you didn't have much control over it...you know, fire some "vertices" at it and a "texture" then the card does everything...that's fast, yes, but it's "inflexible"...it's not "general purpose"...

And these "pixel shaders" are starting to slowly bring that in...basically, when the video card is getting around to plot each pixel, it runs it through your "pixel shader routine" you upload...there are "registers" with the "input values" (screen co-ordinates, texture co-ordinates, etc.) and then you can use some "assembly language instructions" to play around with those "input values" to transform them into whatever "output values" you like...it allows some "user-defined code" to be inserted to give a bit more "control" over what the pixels look like...

At first, it was just a few instructions that could be executed but these limitations are what is getting less and less with each new card and version of the "pixel shader" language...it's getting more and more "flexible" all the time...

> / > *For most games this are just basic attributes.*
> / > *Have you ever tried "TIM" ?*
> / *No. do you have a link to this game. I think you may have mentioned it*
> / *some time in the past, but I couldnt find it sofar.*
>
> *Perhaps it's too old already,*

Re: Can this loop be made faster ?

alt.lang.asm: Re: Can this loop be made faster ?

- > *IIRC the original TIM came on an 760KB floppy during the 286-age.*
- > *It got balls, cannons, ropes, buckets, hamster-motors and a lot more,*
- > *and all the collisions and motions look astonishing real.*

I've not seen that game but it's always a great way to make a game look and feel much better by simply applying things like "inertia", "momentum", "toppling" and little bits of "physics"...when it "moves right" then it just look "real"...even, in fact, if the graphics themselves are all a bit "cartoony", it can still seem "real" enough solely from the way everything moves and bounces and accelerates properly...indeed, if you're making a game and aren't the world's greatest graphics artist then, well, you can "compensate" with programming talent (and thumbing through your maths textbook in the "mechanics" section to find all the proper "formulas" ;) to put in "physics"...because that can make things look much more "real"...

But, as Wannabee says, if you want to see "physics" then "Half-life 2" takes it to another level...indeed, they actually make a "game element" out of it because there's a "gravity gun" weapon, which allows you to fire "force" at things or to pick up objects (a kind of "tractor beam" thing :) and throw them "remotely"...it's quite silly but a lots of fun, just trashing the place throwing objects all over the place...there's a section where you're driving this buggy thing and when you crash, the buggy can topple and land upside-down, with its wheels spinning in the air (the whole game has "physics" on everything so it all spins and bounces with lots of realism :)...and the way you get your car upright again is rather funny...you jump out of the buggy, point your "gravity gun" at it and then fire some "force" at it...this gives it a nice, hard "kick" and you keep firing – bouncing the buggy all over the place (it's quite silly to see this buggy just "bouncing" around all over the screen but that's what makes it all great fun :) – until you manage to bash it back upright onto its wheels...then you can jump back in and carry on driving...

Beth :)