C++ for Game Programming - Love or Distrust? [closed]

Asked 15 years, 7 months ago Modified 3 years, 5 months ago

Viewed 7k times

27

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 4 years ago.

Improve this question

In the name of efficiency in game programming, some programmers do not trust several C++ features. One of my friends claims to understand how game industry works, and would come up with the following remarks:

- Do not use smart pointers. Nobody in games does.
- Exceptions should not be (and is usually not) used in game programming for memory and speed.

How much of these statements are true? C++ features have been designed keeping efficiency in mind. Is that

efficiency not sufficient for game programming? For 97% of game programming?

The C-way-of-thinking still seems to have a good grasp on the game development community. Is this true?

I watched another video of a talk on multi-core programming in GDC 2009. His talk was almost exclusively oriented towards Cell Programming, where DMA transfer is needed before processing (simple pointer access won't work with the SPE of Cell). He discouraged the use of polymorphism as the pointer has to be "re-based" for DMA transfer. How sad. It is like going back to the square one. I don't know if there is an elegant solution to program C++ polymorphism on the Cell. The topic of DMA transfer is esoteric and I do not have much background here.

I agree that C++ has also not been very nice to programmers who want a small language to hack with, and not read stacks of books. Templates have also scared the hell out of debugging. Do you agree that C++ is too much feared by the gaming community?

c++ **performance**

edited May 30, 2013 at 15:28

legends2k

32.8k ● 28 ● 124 ● 237

asked May 6, 2009 at 13:48

amit kumar
21k • 24 • 93 • 128

1 Cell is part of the Playstation 3 hardware, it's CPU i believe. – Skurmedel May 6, 2009 at 13:53

4 Sorry, I don't understand why I am getting negative votes. I hope you see it as a genuine question. – amit kumar May 6, 2009 at 13:59

There are a few questions in here... but there isn't really any conciseness. I think the OP should reformat this in a way that is shorter and clearer. – DevinB May 6, 2009 at 13:59

Apart from everything else, there is no question in the title. If it gets downvoted (I didn't), it's because it's not a good question. I've read it, and I'm still not sure what you're asking, or what I should answer. I'm not even sure if it *is* a question, or a rant/statement. – Stack Overflow is garbage May 6, 2009 at 14:20

15 Answers

Sorted by: Highest score (default)

63

The last game I worked on was Heavenly Sword on the PS3 and that was written in C++, even the cell code. Before that, I did some PS2 games and PC games and they were C++ as well. Non of the projects used smart pointers. Not because of any efficiency issues but because they were generally not needed. Games, especially console games, do not do dynamic memory allocation using the standard memory managers during normal play. If there are dynamic objects (missiles, enemies, etc) then they are usually pre-allocated and re-

used as required. Each type of object would have an upper limit on the number of instances the game can cope with. These upper limits would be defined by the amount of processing required (too many and the game slows to a crawl) or the amount of RAM present (too much and you could start frequently paging to disk which would seriously degrade performance).

Games generally don't use exceptions because, well, games shouldn't have bugs and therefore not be capable of generating exceptions. This is especially true of console games where games are tested by the console manufacturer, although recent platforms like 360 and PS3 do appear to have a few games that can crash. To be honest, I've not read anything online about what the actual cost of having exceptions enabled is. If the cost is incurred only when an exception is thrown then there is no reason not to use them in games, but I don't know for sure and it's probably dependant on the compiler used. Generally, game programmers know when problems can occur that would be handled using an exception in a business application (things like IO and initialisation) and handle them without the use of exceptions (it is possible!).

But then, in the global scale, C++ is slowly decreasing as a language for game development. Flash and Java probably have a much bigger slice of market and they do have exceptions and smart pointers (in the form of managed objects).

As for the Cell pointer access, the problems arise when the code is being DMA'd into the Cell at an arbitrary base addresses. In this instance, any pointers in the code need to be 'fixed up' with the new base address, this includes v-tables, and you don't really want to do this for every object you load into the Cell. If the code is always loaded at a fixed address, then there is never a need to fix-up the pointers. You lose a bit of flexibility though as you're limiting where code can be stored. On a PC, the code never moves during execution so pointer fix-up at runtime is never needed.

I really don't think anyone 'distrusts' C++ features - not trusting the compiler is something else entirely and quite often new, esoteric architectures like the Cell tend to get robust C compilers before C++ ones because a C compiler is much easier to make than a C++ one.

Share Improve this answer

Follow

edited Jul 3, 2012 at 13:48

user142162

answered May 6, 2009 at 14:32

Skizz

71k ● 10 ● 74 ● 109

10 "Games generally don't use exceptions because, well, games shouldn't have bugs" Software in general shouldn't have bugs, but it ends up having. Carefully using exceptions is a good way to handle errors without cluttering the code with a lot of error handling. – piotr May 6, 2009 at 15:11

7 With regards to the above: Console games are tested quite thoroughly by the manufacturer and any failure means not being allowed to publish the game, which means not getting any revenue from sales. So games are generally bug free. The testing covers all known failure modes, such as network disconnect for networked games, controller removal, memory card removal and so on. Out of memory should never happen - (XBox aside) consoles have a known, fixed amount of RAM and it is carefully budgeted to various aspects of the game during development (audio, graphics, AI, etc). – Skizz May 6, 2009 at 16:26

6 @Skizz: Possibly Amit disagrees with the assumption inherent in "games shouldn't have bugs and therefore not be capable of generating exceptions". Some programmers use exceptions in non-bug situations (especially if they're from Java), for example to unwind the stack if your network connection drops deep in some socket-using code. It would be interesting to know why games choose not to do this, even if the answer is just "because we, unlike certain other C++ programmers, prefer to return error codes rather than throw exceptions". – Steve Jessop May 6, 2009 at 17:04

4 Your network connection dropping is exceptional behavior, if you ask me, and a perfectly valid use of exceptions. Its not like you expect this to happen every frame of your game and most good compilers have no speed overheads unless an exception is thrown. – user21037 May 6, 2009 at 17:19

3 Come on. A network connection dropping is not an exceptional behavior. It's normal. – Johan Kotlinski May 15, 2009 at 6:35

Look, most everything you hear *anyone* say about efficiency in programming is magical thinking and superstition. Smart pointers do have a performance cost;

57

especially if you're doing a lot of fancy pointer manipulations in an inner loop, it could make a difference.

Maybe.

But when people *say* things like that, it's usually the result of someone who told them long ago that X was true, without anything but intuition behind it. Now, the Cell/polymorphism issue *sounds* plausible — and I bet it did to the first guy who said it. But I haven't verified it.

You'll hear the very same things said about C++ for operating systems: that it is too slow, that it does things you want to do well, badly.

None the less we built OS/400 (from v3r6 forward) entirely in C++, bare-metal on up, and got a code base that was fast, efficient, and small. It took some work; especially working from bare metal, there are some bootstrapping issues, use of placement new, that kind of thing.

C++ can be a problem just because it's too damn big: I'm rereading Stroustrup's wristbreaker right now, and it's pretty intimidating. But I don't think there's anything inherent that says you can't use C++ in an effective way in game programming.

Share Improve this answer

Follow

answered May 6, 2009 at 13:59

Charlie Martin

112k ● 26 ● 196 ● 264

9 I bet that's the first time anyone's said the words 'game' and 'OS/400' in the same sentence. Kudos for writing such a good OS though. – gbjbaanb May 6, 2009 at 14:18

3 +1 for a really good answer. The only thing to add is to profile in case that something is to slow and then address the "real" culprit found by profiling. Let's not forget what Don Knuth told us: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." – lothar May 6, 2009 at 14:36

9 Quite a number of engineers say that C++ is not for embedded systems. I work as an embedded systems engineer and use C++ whenever possible and the result is higher quality code, which is more compact, easier to maintain and requires less debug. Probably most of those who believe this falacy are those who think C++ C with a few extensions. I have yet to see code with speed problems because it's coded in C++ instead of C. Also, modern C++ compilers do a very good job, and the result is usually as efficient as with C. – piotr May 6, 2009 at 15:18

9 There's a reason the C++ commitee wrote a TR (Technical Report) on the use of C++ in embedded systems - there's too much myth. The TR is available for free from the ISO C++ website. – MSalters May 6, 2009 at 15:27

6 Game developers have very good performance analysis tools. So usually when I hear someone say how smart pointers doing extra work on dereference or virtual calls taking up a lot of time, it's because they saw it in one of their profiling tools. We work with embedded platforms and they all have their own performance idiosyncrasies, so I find it a bit offensive that you assume we're all optimizing our games based on myths and assumptions when you aren't familiar with the platforms we use. – Dan Olson May 14, 2009 at 20:28

12

If you or your friend are really paranoid about performance, then go read the Intel manuals on optimization. Fun.

Otherwise, go for correctness, reliability and maintainability every time. I'd rather have a game that ran a bit slowly than one that crashed. If/when you notice that you have performance issues, PROFILE and then optimize. You will likely find that theres some hotspot piece of code which can possibly be made more efficient by using a more efficient data structure or algorithm. Only bother about these silly little mico-optimization when profiling shows that they're the only way you can get a worthwhile speedup.

So:

1. Write code to be clear and correct
2. Profile
3. **PROFILE**
4. Can you use more efficient data structures or algorithms to speed up the bottleneck?
5. Use micro-optimizations as a last resort and only where profiling showed it would help

PS: A lot of modern C++ compilers provide an exception handling mechanism which adds zero execution overhead UNLESS an exception is thrown. That is, performance is only reduced when an exception is

actually thrown. As long as exceptions are only used for *exceptional circumstances*, then theres no good reason not to use them.

Share Improve this answer Follow

answered May 6, 2009 at 14:26

user21037

7

I saw a post on StackOverflow (that I cannot seem to find anymore, so maybe it wasn't posted here) that looked at the relative cost of exceptions vs. error codes. Too often people look at "code with exceptions" vs. "code without error handling", which is not a fair comparison. If you would use exceptions, then by not using them you have to use something else for the same functionality, and that other thing is usually error return codes. They found that even in a simple example with a single level of function calls (so no need to propagate exceptions far up the call stack), exceptions were faster than error codes in cases where the error situation occurred 0.1% - 0.01% of the time or less, while error codes were faster in the opposite situation.

Similar to the above complaint about measuring exceptions vs. no error handling, people do this sort of error in reasoning even more often with regard to virtual functions. And just like you don't use exceptions as a way to return dynamic types from a function (yes, I know, *all* of your code is exceptional), you don't make functions virtual because you like the way it looks in your syntax

highlighter. You make functions virtual because you need a particular type of behavior, and so you can't say that virtualization is slow unless you compare it with something that has the same action, and generally the replacement is either lots of switch statements or lots of code duplication. Those have performance and memory hits as well.

As for the comment that games don't have bugs and other software does, all I can say to that is that I clearly have not played any games made by their software company. I've surfed on the floor of the elite 4 in Pokemon, gotten stuck inside of a mountain in Oblivion, been killed by Gloams that accidentally combine their mana damage with their hp damage instead of doing them separately in Diablo II, and pushed myself through a closed gate with a big rock to fight Goblins with a bird and a slingshot in Twilight Princess. Software has bugs. Using exceptions doesn't make bug-free software buggy.

The standard library's exception mechanisms have two types of exceptions: `std::runtime_error` and `std::logic_error`. I could see not wanting to use `std::logic_error` (I've used it as a temporary thing to help me test, with the goal of removing it eventually, and I've also left it in as a permanent check). `std::runtime_error`, however, is not a bug. I throw an exception derived from `std::runtime_error` if the server I am connected to sends me invalid data (rule #1 of secure programming: trust no one, even a server that you think you wrote), such as claiming that they are sending

me a message of 12 bytes and then they actually send me 15. In such a situation, there are only two possibilities:

1) I am connected to a malicious server, or

2) My connection to the server is corrupted.

In both of these cases, my response is the same: Disconnect (no matter where I am in the code, because my destructors will clean things up for me), wait a couple of seconds, and try connecting to the server again. I cannot do anything else. I could give absolutely everything an error code (which implies passing everything else by reference, which is a performance hit, and severely clutters code), or I could throw an exception that I catch at a point in my code where I determine which servers to connect to (which will probably be very high up in my code).

Is any of what I mentioned a bug in my code? I don't think so; I think it's accepting that all of the other code I have to interface with is imperfect or malicious, and making sure my code remains performant in the face of such ambiguity.

For smart pointers, again, what is the functionality you are trying to implement? If you need the functionality of smart pointers, then not using smart pointers means rewriting their functionality manually. I think it's pretty obvious why this is a bad idea. However, I rarely use smart pointers in my own code. The only time I really do is if I need to store some polymorphic class in a standard

container (say, `std::map<BattleIds, Battles>` where `Battles` is some base class that is derived from based on the type of battle), in which case I used a `std::unique_ptr`. I believe that one time I used a `std::unique_ptr` in a class to work with some library code. Much of the time that I am using `std::unique_ptr`, it's to make a non-copyable, non-movable type movable. In many cases where you would use a smart pointer, however, it seems like a better idea to just create the object on the stack and remove the pointer from the equation entirely.

In my personal coding, I haven't really found many situations where the "C" version of the code is faster than the "C++" version. In fact, it's generally the opposite. For instance, consider the many examples of `std::sort` vs. `qsort` (a common example used by Bjarne Stroustrup) where `std::sort` clobbers `qsort`, or my recent comparison of `std::copy` vs. `memcpy`, where `std::copy` actually has a slight performance advantage.

Too much of the "C++ feature X is too slow" claims seem to be based on comparing it to not having the functionality. The most performant (in terms of speed and memory) and bug-free code is `int main() {}`, but we write programs to do things. If you need particular functionality, it would be silly not to use the features of the language that give you that functionality. However, you should start by thinking of what you want your program to do, and then find the best way to do it. Obviously you don't want to begin with "I want to write a program that

uses feature X of C++", you want to begin with "I want to write a program that does cool thing Z" and maybe you end up at "...and the best way to implement that is feature X".

Share Improve this answer

Follow

edited May 23, 2017 at 12:25

Community Bot

1 • 1

answered Apr 3, 2012 at 16:29

David Stone

28.7k • 17 • 70 • 86

Just to update this response a little bit: rather than using `std::shared_ptr` for polymorphism in containers, go to `std::unique_ptr` first for a zero-overhead solution.
– David Stone Mar 27, 2013 at 15:07

6

Lots of people make absolute statements about things, because they don't actually *think*. They'd rather just apply a rule, making things more tedious, but requiring less design and forethought. I'd rather have a bit of hard thinking now and then when I'm doing something hairy, and abstract away the tedium, but I guess not everyone thinks that way. Sure, smart pointers have a performance cost. So do exceptions. That just means there **may** be some *small* portions of your code where you shouldn't use them. But you should profile first and make sure that's actually what the problem is.

Disclaimer: I've never done any game programming.

Share Improve this answer

Follow

answered May 6, 2009 at 14:07

Adam Jaskiewicz
11k ● 3 ● 36 ● 37

5

Regarding the Cell architecture: it has an *incoherent* cache. Each SPE has its own local store of 256 KB. The SPEs can only access this memory; any other memory, such as the 512 MB of main memory or the local store of another SPE, has to be accessed with DMA. You perform the DMA manually and copy the memory into your local store by explicitly initiating a DMA transfer. This makes synchronization a huge pain.

Alternatively, you actually *can* access other memory. Main memory and each SPE's local store is mapped to a certain section of the 64-bit virtual address space. If you access data through the right pointers, the DMA happens behind the scenes, and it all looks like one giant shared memory space. The problem? Huge performance hit. Every time you access one of these pointers, the SPE stalls while the DMA occurs. This is slow, and it's not something you want to do in performance-critical code (i.e. a game).

This brings us to Skizz's point about vtables and pointer fixups. If you're blindly copying around vtable pointers between SPEs, you're going to incur a huge performance hit if you don't fix up your pointers, and you're also going to incur a huge performance hit if you *do* fix up your

pointers and download the virtual function code to the SPEs.

Share Improve this answer

Follow

edited May 23, 2017 at 11:54

Community Bot

1 ● 1

answered May 6, 2009 at 14:45

Adam Rosenfield

399k ● 101 ● 522 ● 597

5

I ran across an excellent presentation by Sony called "Pitfalls of Object Oriented Programming". This generation of console hardware has really made a number of people take a second look at the OO aspects of C++ and start asking questions about whether it's really the best way forward.

You can find the presentation here (direct link here). Maybe you'll find the example a bit contrived, but hopefully you'll see that this dislike of highly abstracted object oriented designs isn't always based on myth and superstition.

Share Improve this answer

Follow

edited Dec 26, 2009 at 6:57

answered Dec 26, 2009 at 6:22

Dan Olson

23.3k ● 4 ● 43 ● 56

4

I have written small games in the past with C++ and use C++ currently for other high performance applications. There is no need to use every single C++ feature throughout the whole code base.

Because C++ is (pretty much, minus a few things) a superset of C, you can write C style code where required, while taking advantage of the extra C++ features where appropriate.

Given a decent compiler, C++ can be just as quick as C because you can write "C" code in C++.

And as always, profile the code. Algorithms and memory management generally have a greater impact on performance than using some C++ feature.

Many games also embed Lua or some other scripting language into the game engine, so obviously maximum performance isn't required for every single line of code.

I have never programmed or used a Cell so that may have further restrictions etc.

Share Improve this answer

Follow

answered May 6, 2009 at 15:03

Steven

3,928 ● 3 ● 22 ● 21

4 "Given a decent compiler, C++ can be just as quick as C because you can write "C" code in C++." -- The author of blitz++ library would say "C++ can be faster than C because

you can write C++, the features not in C" – amit kumar Aug 22, 2009 at 10:20

4

C++ is not feared by the gaming community. Having worked on an open-world game engine selling millions, I can say the people in the business are extremely skilled and knowledgable.

The fact that shared_ptr isn't used extensively is partly because there is a real cost to it, but more importantly because ownership isn't very clear. Ownership and resource management is one of the most important and hardest things to get right. Partly because resources are still scarce on console, but also since most difficult bugs tend to be related to unclear resource management (e.g. who and what controls the lifetime of an object). IMHO shared_ptr doesn't help with that the least.

There is an added cost to exception handling, which makes it just not worthwhile. In the final game, no exceptions should be thrown anyway - it's better to just crash than to throw an exception. Plus, it's really hard to ensure exception safety in C++ anyway.

But there are many other parts of C++ that are used extensively in the gaming business. Inside EA, EASTL is an amazing remake of STL that is very adapted for high performance and scarce resources.

Share Improve this answer edited May 14, 2009 at 22:16

Follow

answered May 14, 2009 at 20:21

Johan Kotlinski

25.7k ● 9 ● 79 ● 101

4

There is an old saying about Generals being fully prepared to fight the last war not the next.

Something similar is true about most advice on performance. It usually relates to the software and hardware that was availbale five years ago.

Share Improve this answer

Follow

answered May 15, 2009 at 2:02

James Anderson

27.4k ● 7 ● 54 ● 80

3

Kevin Frei wrote an interesting document, “How much does Exception Handling cost, really?”.

Share Improve this answer

Follow

edited Jul 2, 2021 at 18:47

rbento

11.5k ● 3 ● 65 ● 65

answered May 15, 2009 at 1:54

Luc Hermitte

32.9k ● 7 ● 72 ● 88

It really depends on the type of game too. If it's a processor-light game (like an asteroids clone) or pretty

2

much anything in 2d, you can get away with more. Sure, smart pointers cost more than regular pointers, but if some people are writing games in C# then smart pointers are definitely not going to be a problem. And exceptions not being used in games is probably true, but many people misuse exceptions anyways. Exceptions should only be used for exceptional circumstances..not expected errors.

Share Improve this answer

Follow

answered May 6, 2009 at 14:03

ryeguy

66.8k ● 60 ● 201 ● 263

yeah you prob want to stay away smart pointers and exceptions on really big games. IE AAA titles where there's a lot of code and it has to be as fast as possible (crysis/gow). – Annerajb May 6, 2009 at 14:07

2 And even then, I wish you wouldn't. Too many AAA games are buggy as hell.. Id much prefer the game running a few frames per second slower (and lets face it, micro-optimizations like this will rarely gain you more than a few additional frames) than having the thing leak memory, glitch or crash. Too many AAA games do and I bet its because the programmers worried about shit like this :'(– user21037 May 6, 2009 at 14:30

If you don't expect the errors at all, you cannot write an exception that handles it. If you want a smart pointer that does the same thing as a regular pointer that you allocate with new, use `std::scoped_ptr`. You need to use the right tools for the job, and raw pointers that hold heap allocated memory are rarely (if ever) the right tools for the job. You only use `std::shared_ptr` and `std::weak_ptr` (or their Boost equivalents if you are stuck back on C++03) if you

need to use their functionality, in which case a raw pointer wouldn't work for you. – David Stone Apr 3, 2012 at 16:33

2

I also heard it before I joined the game industry, but something I've found is that the compilers for specialized game hardware are sometimes... subpar. (I've personally only worked with the major consoles, but I'm sure it's even more true for devices like cell phones and the like.) Obviously this isn't really a huge issue if you're developing for PC, where the compilers are tried, true, and abundant in variety, but if you want to develop a game for the Wii, PS3, or X360, guess how many options you have and how well tested they are versus your Windows/Unix compiler of choice.

This isn't to say that the tools are necessarily awful, of course, but they're only guaranteed to work if your code is simple -- in essence, if you program in C. This doesn't mean that you can't use a class or create a RAII-using smart pointer, but the further from that "guaranteed" functionality you get, the shakier the support for the standard becomes. I've personally written a line of code using some templates that compiled for one platform but not on another -- one of them simply didn't support some fringe case in the C++ standard properly.

Some of it is undoubtedly game programmer folklore, but chances are it came from somewhere: Some old compiler unwound the stack strangely when exceptions were thrown, so we don't use exceptions; A certain platform

didn't play with templates well, so we only use them in trivial cases; etc. Unfortunately the problem cases and where they occurred never seem to be written down anywhere (and the cases are frequently esoteric and were a pain to track down when they first occurred), so there's no easy way to verify if it's still an issue or not except to try and hope you don't get hurt as a result. Needless to say, this is easier said than done, so the hesitance continues.

Share Improve this answer

Follow

answered Jul 11, 2009 at 19:31

Tyler Millican

555 ● 4 ● 5

1 +1: People in our industry often forget that trying out new stuff brings uncertainty, which in turn brings risk. If we want to write stable software it is not enough that we can't think of any problems. We have to know by experience that they're not there. – Jørgen Fogh Feb 15, 2011 at 18:25

1

Exception handling is never free, despite some claims to the contrary on here. There is ALWAYS a cost whether it be memory or speed. If it has zero performance cost, there will be a high memory cost. Either way, the method used is totally compiler dependant and, therefore, out of the developers control. Neither method is good for game development since a. the target platform has a finite amount of memory that is often never enough and, therefore, we need total control over, and b. a fixed performance constraint of 30/60Hz. It's OK for a PC app

or tool to slow down momentarily whilst something gets processed but this is absolutely untolerable on a console game. There are physics and graphics systems etc. that depend on a consistent framerate, so any C++ "feature" that could potentially disrupt this - and cannot be controlled by the developer - is a good candidate for being dropped. If C++ exception handling was so good, with little or no performance/memory cost, it would be used in every program, there wouldn't even be an option to disable it. The fact is, it may be a neat and tidy way to write reliable PC application code but is surplus to requirement in game development. It bulks out the executable, costs memory and/or performance and is totally unoptimizable. This is fine for PC dev that have huge instruction caches and the like, but game consoles do not have this luxury. And even if they did, the game dev community would almost certainly rather spend the extra cycles/memory on game related resources than waste it on features of C++ that we don't *need*.

Share Improve this answer

Follow

answered Mar 28, 2011 at 9:18

Alan Chambers

11 ● 1

1 The only error handling that is free in terms of memory and speed is ignoring errors, and let me tell you, for something that's supposed to be free, that sure costs a lot.
– David Stone Apr 3, 2012 at 16:31

Some of this is gaming folklore, and maybe mantras passed down from game developers who were targeting very limited devices (mobile, e.g.) to gamedevs who weren't.

However, a thing to keep in mind about games is that their performance characteristics are dominated by smooth and predictable frame rates. They're not mission-critical software, but they are "FPS-critical" software. A hiccup in frame rates could cause the player to game over in an action game, e.g. As a result, as much as you might find some healthy level of paranoia in mission-critical software about not failing, you can likewise find something similar in gaming about not stuttering and lagging.

A lot of gamedevs I've talked to also don't even like virtual memory and I've seen them try to apply ways to minimize the probability that a page fault could occur at an inconvenient time. In other fields, people might love virtual memory, but games are "FPS-critical". They don't want any kind of weird hiccup or stutter to occur somewhere during gameplay.

So if we start with exceptions, modern implementations of zero-cost EH allow normal execution paths to execute faster than if they were to perform manual branching on error conditions. But they come at the cost that throwing an exception suddenly becomes a much more expensive, "stop the world" kind of event. That kind of "stop the world" thing can be disastrous to a software seeking the

most predictable and smooth frame rates. Of course that's only supposed to be reserved for truly exceptional paths, but a game might prefer to just find reasons not to face exceptional paths since the cost of throwing would be too great in the middle of a game. Graceful recovery is kind of a moot concept if the game has a strong desire to be avoiding facing exceptional paths in the first place.

Games often have this kind of "startup and go" characteristic, where they can potentially do all their file loading and memory allocation and things like that which could fail in advance on loading up the level or starting the game instead of doing things that could fail in the middle of the game. As a result they don't necessarily have that many decentralized code paths that could or should encounter an exception and that also diminishes the benefits of EH since it doesn't become so convenient if there are only a select few areas maximum that might benefit from it.

For similar reasons to EH, gamedevs often dislike garbage collection since it can also have that kind of "stop the world" event which can lead to unpredictable stutters -- the briefest of stutters that might be easy to dismiss in many domains as harmless, but not to gamedevs. As a result they might avoid it outright or seek object pooling just to prevent GC collections from occurring at inopportune times.

Avoiding smart pointers outright seems a bit more extreme to me, but a lot of games can preallocate their

memory in advance or they might use an entity-component system where every component type is stored in a random-access sequence which allows them to be indexed. Smart pointers imply heap allocations and things that own memory at the granular level of a single object (at least unless you use a custom allocator and custom delete function object), and most games might find it in their best interest to avoid such granular heap allocations and instead allocate many things at once in a large container or through a memory pool.

There might be a bit of superstition here but I think some of it is at least justifiable.

edited Jan 9, 2018 at 8:06

answered Jan 9, 2018 at 7:53

user4842163