#Engine Programming is All Plumbing



Amandine Coget Headshot

***Amandine Coget*** *is a French game engine plumber living in Stockholm, freelancing after having worked on the Frostbite and Bitsquid engines. Other interests include crafts, politics, and the politics of crafts.*

(The following is the edited transcription of a conversation we had with Amandine Coget.)

## Introduction

You can probably guess from the name that I was born and raised in France. I also studied there; the degrees I took were programming oriented but not game-specific. It was very different from a Bachelor’s in Computer Science in that half of my studies were computer related (what French calls “Informatique”) but the other half was mathematics, economics, communication, and accounting. That has been extremely useful in my work as a consultant. Ironically, my most useful class has been accounting.

Career-wise, I spent three years at DICE working on *Battlefield 4* and the Frostbite Engine. After that I spent six months at BitSquid[[1]](#footnote-22), and have been freelancing since then. As for my hobbies, there’s weaving which I’m trying to take beyond hobby these days, there’s making my own yarn, and there’s a bit of glassblowing and ceramics.

## The Modes and Pipeline of an Engine

When I was starting out in engine development, the most challenging aspect for me was how much of it is actually related to the pipeline. With small projects, you generally have the data directly in files, like images and such, and then you’ve got the game executable which just loads everything. As soon as you’ve got big projects, though, you will have data that gets processed by a pipeline and then used by the game. So typically you will have a game executable that can run in several modes: it can run as a tool, or it can run as a game. That was initially tricky to wrap my head around: that you had one executable running in different modes and that the code you wrote needed to work in every one of those modes. That’s something you don’t encounter on small projects, because all you need to do there is something as simple as loading the PNGs. For example, if you’re working on mobile, you’re going to want compressed file formats, and you can do that manually by running a standalone tool, or you can have an integrated game pipeline that just processes all the data into compressed formats efficiently. That’s the part that is rarely taught because, again, pipelines are only relevant for large-scale projects.

I would argue the pipeline is always part of the engine because it is something that’s so tightly connected to your runtime—the same team will usually be working on it. The pipeline could be one standard executable that’s compiled with different parameters, or you could pass different command line parameters to your game, but the basic idea is that you don’t just have a “single” exe file. You’ve got those different modes and configurations. It also ties into multi-platform development because when you have so many paths, it takes a mindset shift.

## Onboarding for Programmers

In my experience of onboarding, a specific point of frustration is the lack of up-to-date, complete documentation. This is especially the case for programmers. We usually have resources for content creators because there’s a lot more churn with content creators. Artists, arguably designers, and scripters are the people at a company that change a lot between and during projects. I mean, you know by now that the game industry feeds on blood. As such, programmer onboarding is rarely as developed even though it requires just as much explanation. Programmers new to a codebase need documentation highlighting the company’s established process, and time to find their way around the code.

I remember when a new co-worker joined my team on \_Battlefield 4 \_and was just starting on her first feature. She asked something, so I directed her to someone I thought knew the answer, who directed her to someone else—I think there were like four or five people in the chain—and eventually, the question circled back around to me. You’ve got this problem of knowledge tracking and documentation; figuring out where the knowledge lives. In the end, knowledge management has been the hardest part of onboarding for me. While other industries have entire teams working on compiling knowledge, the games industry typically doesn’t. I think that along with the high rate of turnover where we lose people and therefore knowledge, not having proper processes in place or rarely having tech writers will hurt us. To remedy this issue, it helps to follow good, readable code practices. Comments and documentation can help, but then they have to be kept up to date.

As for new programmers being dropped into this scenario, the big thing is resisting the urge to say “this is crap” early on. The urge to criticize will always be there, and the criticism is often valid. Three months later, though, you get the context and the history, and you’ll understand why things are done that way. In fact, when working on the Customization screen for *Battlefield 4*, which was my big system rewrite, it was a mess but the deeper I went and the more I re-implemented, the more I came to understand why the previous decisions were made. Give up on the idea of writing perfect code—it will not happen, I’m sorry to tell you.

And here is another thing you may not want to hear, but in my experience, after the onboarding, a solid 98% of an engine programmer’s job is plumbing and legacy management. Even if you do write new code, a lot of it will be interfacing with the old.

I think only experience can teach you how to best modify old systems, because as you build up experience, you build up what I like to call the list of bullshit factoids. Just small points about how your actions can backfire, or certain quirks an API has. As you build up this list of factoids, you build an instinct. While I have six years of experience under my belt now, I’m well aware the people who have been doing this for 15 or 20 years will have a much more refined instinct. However, they might also be bogged down by old factoids that aren’t relevant anymore. That’s why it’s good to have a team that’s a blend of ages, experience levels, and backgrounds, because you can have different instincts that can work together.

## Thinking about Usability

Going from working on UI programming to graphics to core systems, the biggest thing I’ve noticed is that it’s become impossible for me to ignore usability. In a way, UI is all the way at the end of the slope, because you will get feature requests from the gameplay team and your work comes at the end of the process. You’re the last one to come in when shit rolls downhill, so you will run into every corner that was cut every step of the way. A part of that work is fetching data from the gameplay code, so if corners have been cut in gameplay code, your UI won’t work. When the gameplay team comes in with last-minute changes, meaning the UI has to be changed, it starts getting messy really quickly.

This reinforces the focus on users, talking to your users, and thinking about usability in general. It’s not as common for programmers on a standalone engine team to talk with designers. But I’ve kept the a habit, even for engine features, to talk with my users a lot. When I was on the rendering team, for instance, I needed to understand if I was on the right track with what was wanted of a feature. Since I was in the DICE offices, I just grabbed an artist, sat them down in front of my computer, and asked them to try to and use what I’d built. This plan really worked because they immediately tried to do something I hadn’t thought about!

Because of that experience, I now regularly ask others if my tools are confusing to use: even if it looked great on paper, what’s the interface like to the user? Your user can be a programmer as well, so you can apply this same UX mindset when designing APIs. I spend a lot of my time “fighting for the users”. I absolutely think a large-ish engine team should always have a dedicated UX designer on hand because it really is a skill. As programmers, we have the nasty tendency to think that we can just figure it out and come up with a decent enough interface on our own. But there are people who study usability for years and know what they’re doing.

## GUI: Immediate vs Retained Modes

When it comes to what form of GUI to use, as always it depends. I haven’t followed how [Casey](https://isetta.io/interviews/CaseyMuratori-interview/)’s take on IMGUI[[2]](#footnote-28) (IM) has evolved over the last few years. For context, while I was working on *Battlefield 4*, what we were doing was shifting the entire UI from using [Scaleform](https://www.autodesk.com/products/scaleform/features/all/gallery-view)[[3]](#footnote-30), which is heavily retained[[4]](#footnote-31), over to something that was more immediate mode and in C++. We rewrote every last bit of UI in the game basically and what we discovered more and more is where IM worked really well and where it broke down. In my opinion, IM is great for simple things, so for debug interfaces, small tools, or demos, it’s a no-brainer. As soon as you need some degree of persistence, like with data-driven systems or allowing designers and artists to customize UI, you will need some kind of extra layer. And that can be something that gets baked down by the pipeline into something that’s more immediate, or it can be something that actually exists at runtime.

But at scale, I find IM code just gets really messy when you’re fetching data from all over the place. This is especially the case if you’re interfacing with a visual scripting system like Frostbite[[5]](#footnote-32) has, which we also call “noodles”. You need some kind of persistent entity just so you can drag one of those noodles into it, and while that gives a lot of power to artists and designers, it gives a lot of headaches to programmers. Noodles are code; they need to be debugged like code, and it turns out that artists are good at making horrendously complicated things really fast. For a project with the scale of the Isetta Engine, where you have only three months, IM is the right choice. For a large AAA project, though, I wouldn’t recommend it as the only approach.

If you can pair IM and retained modes cleanly when designing your system, then it is absolutely a good idea to have both. \_Battlefield 4 \_did a little bit of that because the HUD[[6]](#footnote-33) was very immediate mode-ish and so it had to be fast.

My main work on \_Battlefield 4 \_was the customization screen, which is tons of small widgets controlled by a ton of data. Working on that screen taught me the biggest challenge with UI is fetching all that data. In that case, having persistent entities that just fetch the stuff once and hold on to it can help set up a nice architecture, but it depends on your particular project. Rendering UI can also be challenging because you’re doing layered 2D rendering with a lot of transparency, which modern renderers don’t like. And with UI there will often be more string manipulation than you would like, so you need to optimize that.

## A Brief History of the Frostbite Engine

While Frostbite is known today for accommodating many game genres, it wasn’t designed to do so early on. It had some growing pains. Each new game in Frostbite showed some shortcomings or some things that were missing, and so the engine evolved organically. That shows in the codebase, but I don’t want to harp on Frostbite specifically, what can you do when you’ve got two million lines of code supporting that many games and features?

For a bit of history, Frostbite was originally the *Battlefield* engine, made by DICE in Stockholm. *Medal of Honor: Warfighter,* which we like to forget about, was made on Frostbite as well. Because that game was also a FPS, it was very close to *Battlefield*. The next game to use the engine was \_Army of Two: The Devil’s Cartel, \_which we also like to forget about, which brought third-person co-op. The Frostbite team needed to figure out how to support that gameplay style and what updates needed to be made to the engine as a result.

*Need for Speed: The Run* was the first racing game to use the engine. It brought a lot to the engine tech-wise, because it showed where things cracked. For instance the *Need for Speed* team made a bunch of things for *The Run,* like a road making tool and handling a car’s speed. The vehicles in \_Battlefield \_are done in a certain way, but cars in *Need for Speed* move differently, which affects how to stream in new assets. So each game brought new problems. *Dragon Age: Inquisition* was a massive challenge, because you’re going from an engine designed for FPS titles to a super-massive RPG with conversation systems and more. What I know is that the entire serialization system and the entire saving subsystem had to be rewritten, for example. That’s terrifying to mess with, as you can imagine. Part of that challenge fell to the Frostbite team itself, which at that point was divided between Stockholm and Vancouver, and a lot of it was on the BioWare engine people, who did an amazing job just taking this immense codebase and making tools that they could use to make their game.

## Compartmentalizing Your Knowledge

In a big AAA game engine, you can’t see where things could have been done better just by reading the code because it has over two million lines. It’s so much that you can’t wrap your head around the entire thing. So you will discover what you’re missing as you try to make things, and sometimes you won’t have these realizations until the last second, and you discover that the key connection is missing and that brings us back to plumbing. Once you’re six months into production, you have to make it work. Again, that’s AAA-specific—too big to fail. You can’t know in advance, so you do as much pre-production, as much research, and as much talking with other teams as you can, but in the end you will just go in and see what breaks.

A skill I’ve taken from my time in AAA to my time as a consultant is not needing to understand the entire codebase. When you’re working with this huge, sprawling thing that breaks Intellisense[[7]](#footnote-36), you learn to navigate the code without understanding all of it. You learn to find the part you need to work on and only figure that out, and you accept that you will never understand the whole thing. Focusing on the smaller stuff has been a really precious skill as a consultant because I can just go in your codebase and find my way around in a week or two at most.

## Versioning an Engine

Whether you decide to do versioning inside of your engine or write a tool to change old data really depends on your data and on your users. Can you afford to break retro compatibility[[8]](#footnote-38)? How annoyed do you get at an API that deprecates something you’re relying on? Something like Unity, for instance, is heavily bogged down by having to keep old projects running. Can they afford to break old projects, though? Probably not. If you’re in AAA and you’ve got a dedicated team that has to take the new engine version all at the same time and will have months of work put into this upgrade, you can afford to break old stuff (just don’t do it without telling people, that’s not very nice). Can the conversion be done automatically? Sometimes yes, sometimes no— it depends. Will Perforce’s automatic merging be able to handle changes? Possibly, possibly not. I had the case of a for loop being merged with a while loop so the counter was still being incremented at the end of it, and we only had half the UI rendering. When you are merging tens of thousands of files, you are going to miss that.

This is why you have senior engineers who can guide you on good choices to make, thanks to their experience and instinct. You will have tech directors who call the shots telling you what they’re going for, keeping all the trade-offs in mind. Producers also help with this by keeping you on schedule, because in the end you have to ship.

## Parallelism and Data-Oriented Design

While working on parallelizing Stingray’s data compilation, the biggest challenge for me personally was one very high-profile SDK not being thread-safe[[9]](#footnote-40), meaning that you can only call the “compile asset” function in a single thread; otherwise, the engine breaks down. That’s the problem when you’re parallelizing: if you do it from the start, then you can make sure that everything you’re writing is parallelization friendly, but if you’re working with existing code and doing plumbing you’re going to find all the ways in which it cracks and breaks. Does that happen because it’s using global variables in such a way that it’s not thread-safe at all? Is it some other complex C++ nonsense that makes it thread-unsafe? Parallelization itself isn’t hard per se —though it’s tricky to do well— it’s mostly a challenge if you’re using existing code that’s not thread-safe, parallelization ready, and so on. Data-oriented design can really help with that problem, because when you’ve got tight data, it’s a lot easier to split up the work. But if you’re not doing that, how do you handle things like std::iterators[[10]](#footnote-41)?

## Exposing Data to the Developers

Knowing how to manage memory allocation between subsystems is something that comes from testing, and exposing your engine’s settings to data so the users can change it. Depending on the game’s profile, there will be a lighter or heavier load on different subsystems. So it has to be configured by the user. An example would be if you’ve got a game that’s very effects heavy, your particle system would probably need more memory than it would in a 2D puzzle game that doesn’t have as many particle effects. If you’ve got a game that’s only 2D, your UI system will probably take a heavier load than the 3D system that you’re not using at all.

When it comes to developers abusing that exposed data, that’s where the problem of whether your user is an expert or a neophyte[[11]](#footnote-43) comes in again. Is it something that you can put in documentation, or something that should be impossible to abuse, and cause a crash if used wrong? Is it something that should just be used with caution? Is it a warning, is it an error, etc. When you’re dealing with games in particular, you will generally have users who are focused on shipping and getting the product out the door the way they want it. That’s why I am a strong advocate for engine source access as well, because when you’re doing this last push three-six months before finishing your game, you don’t care about keeping it clean anymore; you just want to get the thing done so the game can be as good as you want it and hopefully release on time. Going back to what it means to abuse an engine, should we assume that as engine developers, we always know best or that in the end it’s our user, the game developer, who should get to make those calls? I still don’t have the answer to this.

## It’s All Plumbing

For a project like the Isetta Engine, aiming at showing what game engine development is like, it’s very good to put the engine together from existing code. Because that’s also true in the professional world: you very rarely get to make features from scratch. It’s all plumbing.

*Interview conducted September 27. 2018.*

1. **BitSquid**, more modernly known as **Autodesk Stingray**, is a discontinued game engine from Stockholm, Sweden. End of sale was announced for January 7, 2018, and afterward it became a plugin for Autodesk 3DS Max known as 3DS Max Interactive. [↑](#footnote-ref-22)
2. **IMGUI** stands for immediate mode GUI which is a code-driven GUI system where on each rendering frame the application needs to issue the draw commands of the GUI (the GUI is not stored for multiple frames) [↑](#footnote-ref-28)
3. **Scaleform** is a vector graphics rendering engine used to display Adobe Flash-based user interfaces and HUDs for video games. [↑](#footnote-ref-30)
4. **Retained GUI** also known as canvas/scene graph, is where GUI is registered once and is displayed, “retained”, on screen until it removes itself from rendering. [↑](#footnote-ref-31)
5. **Frostbite** is EA’s proprietary game engine used across most of their studios. [↑](#footnote-ref-32)
6. HUD stands for **Heads-Up-Display**. It usually refers to overlay on the screen that presents important information to the player [↑](#footnote-ref-33)
7. **Intellisense** is an intelligent code completion feature in Microsoft Visual Studio that is capable of detailing information about the code and objects that the programmer is working with while coding. [↑](#footnote-ref-36)
8. **Retro compatibility,** also known as backwards compatibility, is when a system is setup such that it works with legacy code/input. [↑](#footnote-ref-38)
9. **Thread-safe** code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. [↑](#footnote-ref-40)
10. **std::iterator** is a C++ type that can be used to iterate through collections of elements based on that collection. [↑](#footnote-ref-41)
11. A **neophyte** is a novice to some field, skill, or practice. [↑](#footnote-ref-43)