#Growing Pains in Engine Development



headshot

***Aras Pranckevičius*** *is a Lithuanian programmer who has been working on the Unity game engine since 2006. Before that, he worked on some demoscene demos and small games that you have never heard about.*

(The following is the edited transcription of a conversation we had with Aras Pranckevičius.)

## Experts Are Human Too

When I first started programming, I was mostly just toying around with computer graphics. What made learning engine programming difficult back then was that the Internet was still in its infancy. Even Google didn’t exist yet. My school had limited resources as well, so my only real option was books, which were all written in Russian at the time.

The hardest thing for me to learn specifically was how to program on a team. When other people are thrown into the mix, all of a sudden everything changes. You no longer do everything by yourself and you have to somehow collaborate. As a complete introvert, communicating with people was one of the biggest challenges I faced. Learning how to work in a team is a super-valuable skill that the universities I’ve seen don’t talk about a lot, or maybe not at all. As you work on your shared code base and your engine ideas that you have, teamwork is extremely useful. It is hard in some places, but at the end of the day there’s only so much one programmer can do.

Also, in many of [my blog posts](https://aras-p.info/blog/), like my recent path tracer[[1]](#footnote-23) [series of blogs](https://aras-p.info/blog/2018/03/28/Daily-Pathtracer-Part-0-Intro/), I write about topics where I still do not completely understand them—even as a professional with roughly 15 years of experience. People have told me that admitting I have no idea what I’m doing is a refreshing thing to see, because there’s this expectation that industry veterans understand everything. Since we’re all human, that’s obviously not the case!

## Dingus: An Engine to Forget

Back when I made things for the demoscene[[2]](#footnote-26), I worked on a game engine called Dingus with a few others. I don’t think the engine had any special architecture or technology; it was just a bunch of code that we found useful. Back in around 2002 or 2003, the only engines to come with actual tools were RenderWare[[3]](#footnote-27) and Unreal Engine 2. Engines at the time didn’t come out of the box with any decent tools. An engine was basically just a bunch of code, and there was nothing in there for artists. So our “engine” was just a bunch of C++ code that we used in our demos, and the only tools we had were mesh exporters from 3ds Max[[4]](#footnote-28). These days, you just export any FBX[[5]](#footnote-29) or glTF[[6]](#footnote-30) file and there are ready-to-make libraries to read that, which was not the case back then.

To be fair, I don’t think I had any particularly clever insights when writing Dingus that would help me in the future. What it helped me with in my career, though, is that it was the main reason why I got hired at Unity! The Unity founders told me the reason why I got hired was because I was writing some some messages to a mailing list about physics engines. In particular, it was about a physics engine called the Open Dynamics Engine[[7]](#footnote-31), which Unity used before it moved on to PhysX. The Unity founders were reading this mailing list, and they saw my messages. I guess they thought I was not I was not totally stupid, because I ended up getting the job! It also helped immensely that I had my own website with a blog and tech demos. That said, I don’t think the actual C++ code I was writing for Dingus back then was useful in the end. *If you are looking to get hired today as an engine programmer, I think making content like blog posts and videos about your work will be more useful than your actual code.*

## Reflecting on Windows Unity Editor and Graphics Abstraction

Although we had to shuffle a lot of code around to make the Unity editor for Windows, there weren’t many decisions that I made that I regret. Unity started as Mac-only software in 2004 or 2005. Back then, actually, Macs were not the hip thing they’ve become in the last 10 years; this was before iPhone existed. At the time, almost nobody had a Mac. Despite this, for some reason, Unity started on a Mac and remained that way for a long time. When the team realized the majority of game developers are not on Mac, we knew we needed to make a Windows version. This was a huge undertaking because Unity’s editor tools were written with a lot of Mac-only assumptions. It was essentially Cocoa[[8]](#footnote-33) for the UI and the various UNIX[[9]](#footnote-34) assumptions about files. The most painful aspect of porting to Windows was the asset pipeline in particular. On a Mac, for example, there’s no such thing as an application having exclusive access to a file. If some process is reading a file, you can virtually delete it, and then when that process goes away, the file actually gets deleted. On Windows, if a file is being used, another process cannot just go and delete it. Stuff like that was probably the most annoying to get through, as well as the differences between how Cocoa and Windows UI.

There were definitely some decisions I made back then that I regret. Since Unity started on a Mac, the engine had been OpenGL[[10]](#footnote-35) only. Neither Metal[[11]](#footnote-36) nor any other alternatives existed at the time, so OpenGL was our only option. From there, we started to add Direct3D[[12]](#footnote-37) 9 (DX9) support, and so we made a little abstraction layer for the graphics API. Since we were doing this in around 2006, shaders already existed but more complex elements—like the concept of compute shaders[[13]](#footnote-38)— didn’t exist at all. Our abstraction layer for the graphics looked like a very DX9-style API, which we later modified when adding Direct3D 11 (DX11) and Playstation 3 compatibility. It stayed in this sort of legacy DX9/DX11-style API for a very long time because didn’t do enough internal refactoring. Right now, a bunch of people at Unity are doing that, but, for example, getting DX12 working with this DX11-style abstraction was very painful.

There probably isn’t a good way to abstract out a single system. You don’t know what or how to abstract until you have two different ways that you need to do some particular systems. Or, in terms of graphics, until you get two or three graphics API’s that you need to abstract. For example, our abstraction for DX9 at the time was okay, but we neglected to keep modernizing. Currently we are doing that, but we were several years too late that made catching up a very painful process. Modifying engine architecture on software as big as Unity is painful. Some aspects of it are insanely hard to change. I saw a [tweet](https://twitter.com/mcclure111/status/954137509843398656) many months ago that said:

Library design is this: You have made a mistake. It is too late to fix it. There is production code depending on the mistake working exactly the way the mistake works. You will never be able to fix it. You will never be able to fix anything. You wrote this code nine seconds ago.

It’s not exactly true that you will never be able to change it, but some of the decisions that we made in the engine are near-impossible to change without adding a parallel system that, for some amount of time, lives right next to the old one, that it has to replace. Hopefully the new system is better than the old one and people move toward the new one, and then maybe, eventually, you can remove the old one.

As we speak, I think some of the components in Unity that are used for the in-game UI system are from three UI-system generations in the past. Right now, we have in-game UI, and then before that we had something like IMGUI-based[[14]](#footnote-40) in-game UI, and before that there was the GUI text component from Unity 1.0. I think we are removing those right now, so *it only took about 12 years*!

## API Responsibility

How much the API protects the developer depends on who your target audience is. At Unity, we have exactly this problem; a large percentage of Unity users are not very experienced developers, so they need something that’s easy and robust. At the same time, we want to have content that pushes the limits of hardware to serve people who actually know what they’re doing. For the experienced programmers, the API and the system shouldn’t get in their way.

Admittedly, we are not always great at this. The best approach I’ve seen overall is where you have two levels of API’s. One would be a low-level API that is super efficient, super explicit, and doesn’t protect you from anything. An example of this would be Vulkan[[15]](#footnote-42), or DX12, both of which you have to be an expert to use. In addition to the low-level API, you should have another API that’s easy to use, even if it’s not 100% efficient. For the other 6 billion of us, it gets the job done. We’re trying to bring this low-level API and high level API split into our systems at Unity, but we’re not quite there yet. If you’re making your first engine as a learning project, you don’t need to worry about this, but it could be valuable for your next engine.

There are some ways to handle adding new features to an engine without breaking production code, one of which is having a package manager[[16]](#footnote-43). We just shipped the Unity with package support a year ago, so not everyone in the Unity ecosystem has moved to that yet. We’ll see how that goes.

Conceptually, package management is different from engine versioning in that you could upgrade parts of the engine that you care about. For example, if there’s a new feature in the physics system that you really really want, but you want to stay on the current rendering engine, you could just upgrade the physics system. In a perfect world, physics would be a separate package without dependencies to anything and you could just mix and match. In that case, you could upgrade physics and keep everything else the same—whether that will actually work in practice, we’ll see. Right now, there’s a whole bunch of the Unity engine that is not put into packages yet. To some extent, we want to get everything packaged.

Getting modules to talk to each other, however, is not a trivial problem. I think the only way to design interfaces that’s actually stood the test of time was not to design them in terms of function calls or classes, but to design them in terms of data formats. If you get a TGA image file, which has been around for 30 years in its simplest form, it’s just specifications of how the data is layered. And then, you basically don’t care how the TGA writer/reader is implemented. I think one way of making systems communicate is to define the data protocol between them. They could communicate through a shared memory space or a socket[[17]](#footnote-44), and then you don’t care whether the class changes that is writing the data or if they’re even the same language.

However, there’s a bunch of current functionality that we don’t plan to package. That’s especially true for the systems that we are about to replace. For example, we made the [Scriptable Render Pipeline](https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html) (SRP), where users can write their own graphics pipeline in a high-level C# API. We don’t intend to put our legacy pipelines into separate packages anytime soon, because we expect that people will move on to the new pipelines anyway.

Our approach with the SRP was to make the actual API where you express your rendering pipeline high-level enough that you never operate on a single object. Instead, you operate on sets of objects. For example, there could be an API call that does culling. It doesn’t return a list of visible objects; it returns some handle to the whole set of visible objects that you can only do partial queries on. With that, you could search for everything that’s visible or everything that’s opaque, but you don’t have to iterate over each object and do decisions on each of those. Conceptually, the API works on sets of objects, sets of lights, etc.

We also weren’t exactly making a switch to a less efficient or tightly-pathed system. As our previous non-malleable rendering pipeline grew over ten or twelve years, it gained so many hidden decisions and branches to handle various feature interactions that it was no longer super tight and efficient. It was now being used to handle various corner cases that only happen in rare cases.

## From Graphics to Plumbing

Recently in my career at Unity, I changed roles to be a dev tools engineer. Switching from graphics to build tools didn’t impact my perception of game development very much, and I think that’s because I was already dabbling in non-graphics work during Unity’s early days. Even during my time in graphics, once in a while I would be doing something else, so I already had a good overview of various systems outside of graphics.

What I didn’t fully understand was the differences in machine configurations when people build code. You would expect everyone who uses Windows gets the Windows version of software. It turns out, though, there’s about ten people who are on Windows but run everything from a Cygwin[[18]](#footnote-47) shell, and the Cygwin shell pretends it’s Linux. Some people check out their source code into a folder that is over 100 characters long, and Windows, to this day, still has a maximum path length of 260 characters. Someone else might have a Windows that’s localized in French, so all their error messages are in French, which means your tools cannot parse the error messages and expect something understandable. Conceptually you might understand these various exceptions, but you don’t realize how much of a hassle all that is until you have to [deal with that](https://www.google.com/url?q=https://media.giphy.com/media/2ifaifs54xq7hKoUBt/giphy.gif&sa=D&ust=1540510365419000&usg=AFQjCNGlNezwE03C5WoBGVfnrxh1YFWzmw). My advice would be to get your team onto a setup that is as uniform as possible. Not having to worry about tech differences makes things so much easier.

I watched part of the Isetta team’s [talk with Casey Muratori](https://isetta.io/interviews/CaseyMuratori-interview/), and in one part he brought up the question of who “build engineers” are. Actually, that’s what I am! I guess Casey comes from a different setup, because at Unity we have five hundred engineers writing code on the same codebase. Now, you could argue whether that’s a good thing or a bad thing, or whether you should have five hundred engineers in the first place. With that many engineers at work, I think having at least two or three build engineers actually helps; whatever you can do to make your programmers’ lives more productive or less annoying is a useful thing.

Part of that work has been profiling our code, which can be very useful. However, if you add profiling capture that no one will ever look or do something about it, it’s kind of pointless, right? What I have noticed, and especially in the in the build area, is that you have to make profile information really visible. For example, right now in Unity’s build system, what we do is that as each C++ file is being compiled, it prints the time it took right to next to the file name. The numbers are nicely aligned, too, so that if it takes two digits of seconds, it’s easy to spot. Just adding that was a super easy thing to do, and that makes people who wouldn’t ordinarily pay attention to build times take notice and address the problems they find.

## Modularity is the Future, Maybe?

I don’t know what the future holds for game engines. Looking back to when we were just getting started with Unity, I remember some people thought it was a stupid idea because nothing we could make would be able to compete with Unreal. That is still true to an extent in the AAA game space, where aside from in-house engines, Unreal Engine 4 is the strongest option today. Knowing this, we tried for a long time not to compete with Unreal Engine; we positioned ourselves as an indie web/mobile engine (for better or worse). I think if we were to compete with Unreal from Day 1, we wouldn’t have survived as a company.

That said, I see the future of game engines being more modular. That’s what we are currently trying to do with Unity itself in terms of packages and and modules, but the risk is going overboard with modularity. For everything that’s good about JavaScript npm[[19]](#footnote-51), they sometimes go too crazy; they have a single line of code becoming its own module. As Unity and Unreal are currently these big, monolithic pieces of tools and functionality, they aren’t inherently versatile enough for different game genres. While they both have some malleability, you still have to deal with gigabytes of extra stuff. I think something more modular will happen in the future – I can hope that will be Unity, but we’ll see.

*Interview conducted on October 8, 2018.*

1. **Path tracing** is a realistic lighting algorithm that simulates light bouncing around a scene. It uses the Monte Carlo method to give a faithful rendition of the global illumination of the scene. [↑](#footnote-ref-23)
2. The **demoscene** is an international computer art subculture focused on producing demos, which are self-contained, sometimes extremely small, audio-visual computer programs. [↑](#footnote-ref-26)
3. **RenderWare** is a game engine by Criterion Software that launched in 1993 and continued to regularly support games through 2010. It was known for providing an off-the-shelf solution to the difficulties of PS2 graphics programming. [↑](#footnote-ref-27)
4. **Autodesk 3ds Max**, formerly 3D Studio and 3D Studio Max, is a professional 3D computer graphics program for making 3D animations, models, games, and images. [↑](#footnote-ref-28)
5. **FBX** is a proprietary file format owned by Autodesk that is mostly commonly used for 3D model and animation data within the games industry. [↑](#footnote-ref-29)
6. **glTF** (or GL Transmission Format) is a royalty-free file format for 3D scenes and models using the JSON standard. [↑](#footnote-ref-30)
7. **Open Dynamic Engine**, or **ODE**, is a free and open source physics engine written in C/C++ that can do both rigid body dynamics simulation and collision detection. [↑](#footnote-ref-31)
8. **Cocoa** is Apple’s native object-oriented API for macOS. [↑](#footnote-ref-33)
9. **UNIX** is a family of multitasking, multiuser operating systems that derive from the original AT&T Unix, originally developed at Ken Thompson, Dennis Ritchie, and others at Bell Labs. It’s main comparable is Microsoft’s DOS, which is mono-task and monouser. [↑](#footnote-ref-34)
10. **OpenGL**, short for Open Graphics Library, a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. It’s the underlying rendering library for many modern game engines. [↑](#footnote-ref-35)
11. **Metal** is a low-level, low-overhead hardware-accelerated 3D graphic and compute shader API developed by Apple and debuted in iOS 8. It combines functions similar to OpenGL and OpenCL under one API. [↑](#footnote-ref-36)
12. **Direct3D** is a graphics API within Microsoft DirectX used for creating windows and rendering, and serves similar purposes as OpenGL. [↑](#footnote-ref-37)
13. A **compute shader** is a shader stage in the graphics rendering pipeline that is used entirely for computing arbitrary information. It is typically used for tasks unrelated to rendering. [↑](#footnote-ref-38)
14. **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-40)
15. **Vulkan** is a low-overhead, cross-platform 3D graphics and compute API that targets high-performance realtime 3D graphics, such as video games. [↑](#footnote-ref-42)
16. A **package manager** is a system which handles the installing, updating, configuring, and removing of a collection of software libraries. [↑](#footnote-ref-43)
17. A **socket** is an internal endpoint for sending or receiving data within a node on a computer network. [↑](#footnote-ref-44)
18. **Cygwin** is a Unix-like environment and command-line interface for Microsoft Windows. It provides native integration of Windows-based applications and resources with those of a Unix-like environment. [↑](#footnote-ref-47)
19. **npm** is a package manager for the JavaScript programming language. It’s the default package manager for the JavaScript runtime environment Node.js. [↑](#footnote-ref-51)