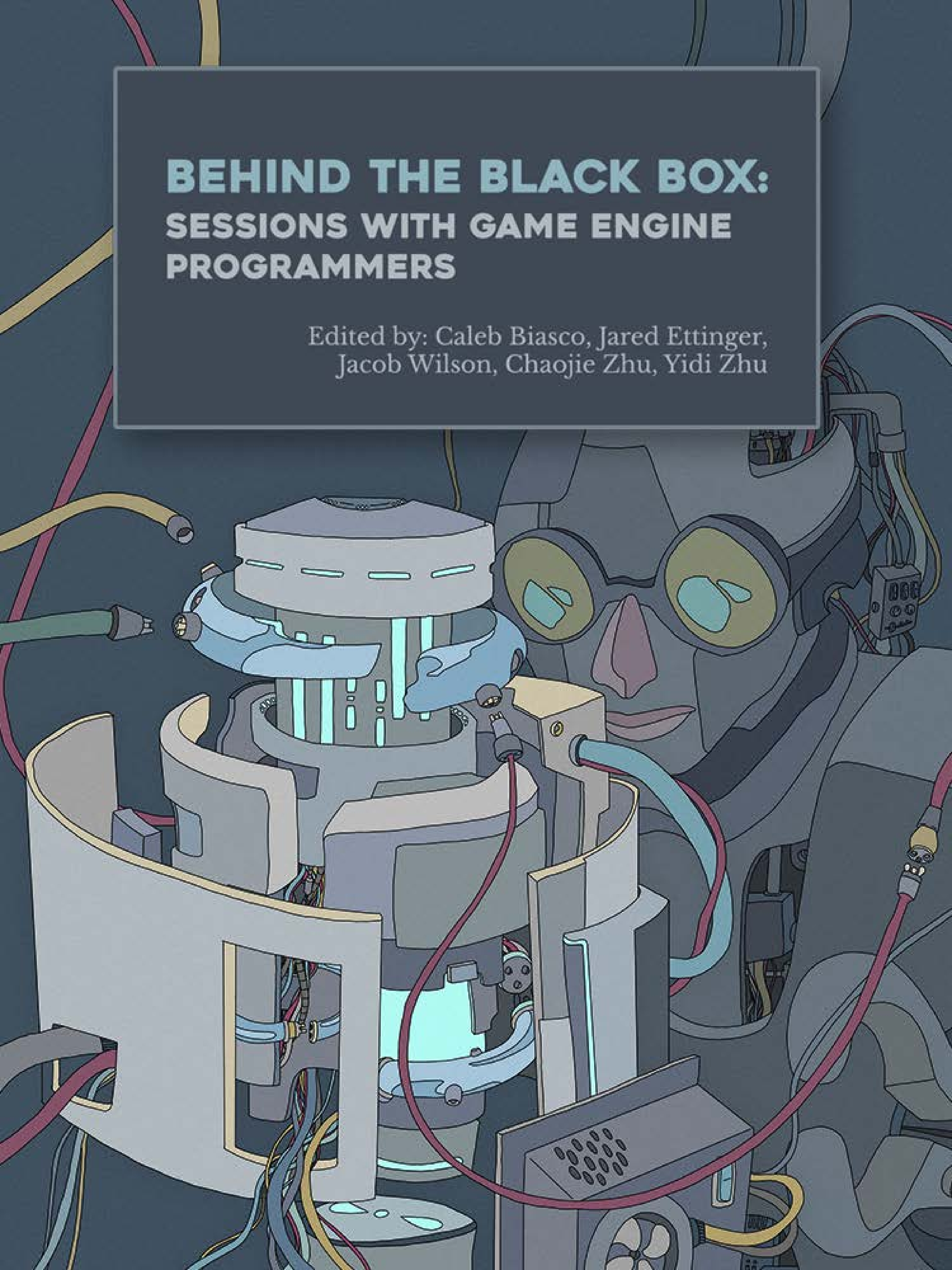

BEHIND THE BLACK BOX:
SESSIONS WITH GAME ENGINE PROGRAMMERS
Edited by: Caleb Biasco, Jared Ettinger,
Jacob Wilson, Chaojie Zhu, Yidi Zhu

Behind the Black Box

Behind the Black Box

Sessions with Game Engine Professionals

Caleb Biasco, Jared Ettinger, Jacob Wilson,
Chaojie Zhu, and Yidi Zhu

Carnegie Mellon University: ETC Press: Student

Pittsburgh, PA

Behind the Black Box by Carnegie Mellon University: ETC Press: Student is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, except where otherwise noted.

Copyright © ETC Press 2018 http://press.etc.cmu.edu/

TEXT: The text of this work is licensed under a Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) (https://creativecommons.org/licenses/by-nc-nd/4.0/)

IMAGES: All images appearing in this work are licensed under a Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) (https://creativecommons.org/licenses/by-nc-nd/4.0/)

Contents

Preface vii

Foreword ix
Cort Stratton

Why Game Engine Development is Worth Learning 1
Casey Muratori

An Engine Developer's Toolbox 14
Jeff Preshing

Engine Programming is All Plumbing 25
Amandine Coget

The Definition and Beginning of a Game Engine 35
Adam Serdar

Growing Pains in Engine Development 44
Aras Pranckevičius

Wisdom from Working at AAA Studios for 15 Years 54
Elan Ruskin

Going Beyond the Books 65
Shanee Nishry

Thinking About the Data 78
Martin Middleton

The Engine Sandwich: Made with Super Meat 88
Tommy Refenes

A Lost Art 98
Raymond Graham

About the Team 109

About the Publisher 110

Acknowledgements 113

Preface

We don't know what we're doing. And how could we? This project started when our team of five noticed a significant gap in the field of game engine programming; there was no clear starting point for novices. We found many completed engines with demos of their technical feats, but we struggled to find a timeline, decision breakdown, or process behind those engines. No matter where we looked online, the sure-fire responses others would give were to read a textbook that is over one thousand pages long — or to do it yourself. We thought if we were to do the latter, we could document the down-and-dirty process and leave something for other aspiring engine devs to discover.

The idea was proposed to our graduate faculty as a self-driven project to develop an engine in three months. And being that we were students and (more importantly) novices to the field of engine development with little to no prior assumptions, we would be able to document and speak on the topic with a unique perspective. Unique, however, doesn't always mean useful! Our conclusion was to interview professional game engine programmers alongside our engine development. In this way, we paired our novice understanding with industry experts on the fundamentals of game engine programming

For most of the interviews you are going to read, we had no prior connections to the interviewees. We happened to think they were doing interesting work, so reached out to them via public email/social media and hoped for the best. Luckily, these individuals thought our work was interesting as well, and they were happy to help us. We interviewed most of them over Skype and some in person on a trip to Los Angeles. Each interview has

been transcribed and edited for clarity, with approval from the interviewee.

From these interviews, we came to understand more of how this rarely discussed field works. There were many times when the team would be hemming and hawing over different approaches for an engine feature, only to have an interviewee incidentally bring up something that illuminated a better approach to us. If you are developing a game engine yourself, we think you will get the same kinds of “a-ha!” moments from their stories. If you are just interested in learning more about the game industry, then you will find a behind-the-scenes look from a diverse array of perspectives.

Foreword

Speaking as a former aspiring engine programmer, I hope you realize how lucky you are to be holding this book.

When I was an undergraduate at Carnegie Mellon's School of Computer Science in the late 1990s, I (along with many young programmers, then & now) was intensely interested in game development. I knew that games were built on top of something called an "engine", but at the time, a "game engine" was something that Actual Companies with Actual Budgets paid bags full of Actual Money to get access to. Both the engines themselves and the development that went into them were closely-guarded trade secrets, with the possible exception of occasional nuggets slipped into John Carmack's .plan files (hey, remember .plan files?). I had no shortage of passion for the material, but as a starving college student with embarrassingly few sackfuls of money to my name, I felt that I had no choice but to figure everything out on my own. And so, teeth firmly clenched, I proceeded to dig my way down to the very bottom of the technology stack, thus beginning a life-long journey to claw my way back up to actual game development.

Twenty years later, I really can't complain about where I ended up. I've been fortunate enough to work with amazing teams teams at Electronic Arts, Sony, Google, and Unity, and have contributed to thousands of games at every scale, from giant AAA blockbusters to the humblest indie offerings. I certainly know more about engine programming than I did back in college. And yet, reading through these collected interviews, I think of all the mistakes and false starts I could have avoided along the way, and I can't help but feel a bit jealous of the head start these students will have vs. myself at their age.

Reading Casey Muratori's interview, I'm painfully reminded of the months I've wasted over-thinking the design and feature set of APIs without a single well-considered use case. Reading Amandine Coget's, I lament the weeks I've lost to indecision paralysis, unwilling to make forward progress on a change until I fully understood the entire labyrinthine context of the system. Through Jeff Preshing and Aras Pranckevičius, I recall (with appropriate shame) the years I spent dismissing interpersonal communication and professionalism as the sort of skills only "the suits" needed to worry about. Alongside Raymond Graham, I remember the PlayStation 3's SPU co-processors, and… well, actually, I have no regrets there are all; SPU programming was incredible, and if you didn't get to experience it, you seriously missed out on something special. You have my sympathy.

The remaining interviews are equally illuminating. However, the most important takeaway from this book may not be anything in the interviews (or the foreword, but that's an excellent guess as well). The very existence of the book itself is a blessing, as is the simple fact that these distinguished developers have made themselves and their wisdom available to fellow professionals, students, and hobbyists alike. Game engine programming is a dark art, it's true, and its practitioners are a rare breed. But it turns out they're also a generous bunch. Today's passionate developers don't need to figure things out on their own; whether you're a member of a giant team working on the next great AAA engine or an inquisitive individual, the tricks of the game engine trade are often only a tweet away. (Hey, remember Twitter?)

Meanwhile, complete professional-grade engines like Unity and Unreal are now available for anybody to play with (no bags of money required!), and a sprawling ecosystem of open-source libraries gives programmers an unprecedented peek under the hood of their favorite tools. It's never been easier to get started as a game developer, and I believe it's only going to get easier going forward. But even the best software (and its source code) only shows you the final product, not the thought processes that created it. Tools give you the what, source code shows you the how, and a book like this one exposes the why.

I commend the developers for taking the time to share their knowledge. I applaud the members of the Isetta team for collecting and publishing that knowledge. And I salute you, the reader, for being curious enough to pursue it. Speaking as a professional aspiring engine programmer, I can't wait to see what you build with it!

Cort Stratton
Senior Software Engineer, Unity Technologies
Los Angeles, California
November 2018

Why Game Engine Development is Worth Learning

Casey Muratori

__Casey Muratori__ is the lead programmer on 1935*, an upcoming interactive story engine project, and the host of* Handmade Hero*, an instructional series for game engine programmers. His past projects include* The Witness*, the Bink 2 video codec, and the Granny Character Animation System.*

The Shortage of Engine Programmers

I won't pretend I have the data to support that there is a real shortage of engine programmers, but if today you told me I needed to staff up a modest ten-person engine team, I would have almost no idea where to get eight of those people. And the two that I maybe *do* know where to get, I'd have to hire from some other team. So I'm strictly talking about this from personal experience in the game industry. It is an anecdote, not a statistic, when I say I just don't find that people know where to hire engine programmers. I know plenty of people who have hired people who they really don't think are even that good at engine programming, but who were definitely the best they could get.

Similarly, I know plenty of people who would hire another engine programmer right now if they could find one, but there's no one who meets their cutoff. Some of that is — especially on smaller teams — because you can't afford to hire lots of junior people. Maybe there's some raw talent out there and it just needs to be developed. Data-wise, I don't want to make claims about this that I can't substantiate, but real engine programming experts are very hard to find, and I can't offer you a reason for that. I just assume the reason is that even something as simple as using Unity kind of requires you to understand 3D math at a pretty decent level if you're not gonna ship one of these games with tons of obvious bugs.

Being a game engine programmer is just a hard profession. It demands that you know a lot of things, so it's not for the faint of heart. It requires a lot of expertise, a lot of time spent learning even to just be a competent engine-side programmer. You need a lot of skill to modify Unreal Engine in a small way, or even to competently program a motion controller. These are just skills that people don't seem to have in abundance. Even today when you can just go get an engine so you don't need to build it from scratch, we still haven't gotten to the point where we have enough people with expertise.

We don't have enough expertise versus how much we could be using to make games better. I suspect if you talked to other people in the game industry, they would have similar opinions about how hard it is to find good engine programmers. I don't want to put words in their mouth, but it's not the kind of thing that you can just go find someone to do. I think we're getting better at pipelines for finding artists thanks to social media and online tutorials about what goes into making a game asset. Perhaps also because of school curricula that was not around in the old days.

What I have not seen change is the number of engine programmers. The last three engine programmer hires I know about — or was in any way involved with — all came from *Handmade Hero*. If it wasn't for *Handmade Hero*, who knows if they even would have done it? That was the reason for *Handmade Hero*:

because I believe this way of life is worth preserving. It's talking specifically about a practice and a discipline and a mental model. I never said I want to show you how I make a game. I said "this way of life" specifically. I feel like making a game is covered well enough elsewhere.

The Confusion with Game Engines

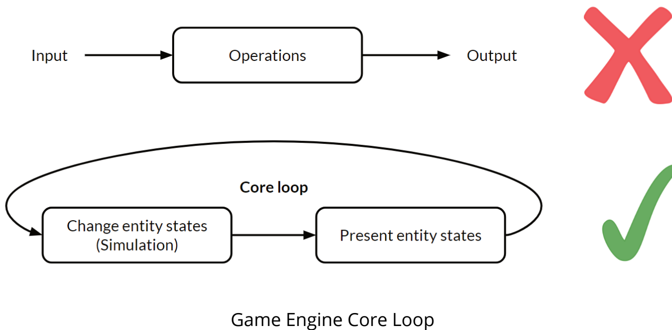
Problem 1: An Amorphous System

There are two fundamental problems in game engine development that I think set it apart from a lot of other types of development. The first one is that it is not a stateable problem in any way. Half of everything that programming does in the past or today looks a lot like stdin/stdout[1]. You can phrase it as "here are the set of inputs, and here are the set of outputs that come out of it." So most of the things that programmers are used to thinking about are "I'm working on deep learning[2]. Here's a set of input images and a set of output tuned neural net parameters. How do I make the best translation between these two things?" Or, "I'm a natural language processing person, here's all the corpus[3] I want in and here are the noun tags I want out, or the sentence tags I want."

I think one of the biggest challenges for an early game engine programmer is making the leap from input-output thinking to this amorphous system. It's very confusing how that happens at first, because even though you're not necessarily aware of it, everything you've ever done prior to that looks a lot more like this input-output phrasing. One of the really good things that you can do at first is to try to figure out the "core loop" of a game engine; the golden differentiator of a simulation. In a flight simulator or a game, they look like this real-time loop where I have a set of entities (each with their own states) and I go through a simulation to change those states. That's a certain process. I then

1. **stdin** and **stdout** are the functions that handle program input/output in the standard library of the C programming language.
2. **Deep learning** is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. Deep learning is a key technology behind driverless cars, enabling them to recognize a stop sign or to distinguish a pedestrian from a lamppost.
3. In linguistics, a **corpus** or text corpus is a large and structured set of texts.

go through a way of presenting those things, and then I return to the beginning.



Game Engine Core Loop

Just getting yourself comfortable with that and the fact that everything you build has to build out of that is just the first big step. It's not input-output anymore. It doesn't look like a web program; it doesn't look like the neural network; it doesn't look like a parser; doesn't look like a server; doesn't look at any of these things. It looks like that — a "core loop".

Problem 2: The Complexity Explosion

The other thing is the complexity explosion. So again, it's not input to output where things are very clean. You now have all of these systems that are all happening at the same time. They all interact, they all overlay on top of each other. The physics and the rendering and the blob, etc. All this stuff comes together.

You really need to be able to do discoverable architecture in order to do these things correctly. Most people are just not familiar with that process — I called it Compression-Oriented Programming in the past. I would define that as the technique of starting with the simplest way possible. I think this is Jonathan Blow's term that he uses often: Write whatever is obvious to you. Don't even think about the rest of the game engine. Focus on the one thing you need to do at the time. Think about it more like the standard in, standard out (stdin, stdout) that we're all used to. How do I it to just get the input and the output; to do the thing I need to do in isolation? No code design, no cleanliness, no noth-

ing. You need to be able to clear everything from your head and just do that, and then pull that out into the architecture.

One of the biggest mistakes I think even experienced game programmers will make is they go the other direction. They try to start with the architecture and drill down to the thing they want to do. Never in my entire life have I seen a good result from that. It almost always has to be "refactored" at the end. That's because when you start from a conception of how something will plug together, you don't see all the details. You don't know all the things that you're gonna have to do when you go to solve the problem, so you always forget some of the details. On the other hand, if you just implement the real thing first and then pull it up into the architecture, you almost always end up with better results. I like to think about those problems.

The "Boundary Value Problem" Architecture Method

There's this thing called boundary value problems[4] in mathematics, and there are different ways of solving them. There's the shooting method: I start here, I see where I end up. There's also the method of solving backward, where I look from the endpoint and try to see what I could do differently. Building up a skill set of how you work on code that allows you to work from either side of the problem ends up being really valuable. Like I said, you usually want to start with an isolated solution and loft it up into the architecture. But then, when you get to a certain level of that lofting you want to think from the other side down. You want to determine what a good API for this should look like, what are some things you maybe didn't think about while implementing it. And things that you should maybe change, that wouldn't break the algorithm, but that would let it work better with this integration.

You almost want a ping-pong development process where you start at the solution, work backwards a little, go to the architecture side, work forwards a little, and come back. The best programmers at doing this

4. A **boundary value problem** in mathematics is a problem whose solution that satisfies boundary conditions that act as constraints for the solution. More can be learned on its Wikipedia page.

sort of thing are able to make it fit really nicely at the end, so both sides can be happy. I'd like to give a shout out to Allen Webster[5] for that, he works at RAD Game Tools now. I've talked about those two things separately before, and he pointed out that I need to connect them.

Competence, Coding Style, and Working in Teams

Casey's Definition for a Competent Programmer

Part of developing a good programming style has very little to do with things that people normally focus on, which are minor things like "Did I overload operator=[6] to prevent a copy". They focus on all of these rules that they never even tested. They ignore the most important thing which, in my opinion, is how easy is it to read your own code later and know exactly what it does. This is also why I tend to not comment code until it's "done done," because I find that the comments end up being out of date and counterproductive. Since the comments are describing the thing that was before the latest one, it's actually worse than having a no-comment.

So that's I think what keeps *Handmade Hero* so flexible and easy for me: To not have to worry when I come back to it on a weekend to start doing a stream. I know part of being a competent programmer is writing code in a way that doesn't require me to keep it all in my head. I can easily go look at the function names and I just know what they do.

This is doubly easy if I'm the programmer, because I know what assumptions I make and if I just always make those same assumptions then I know I don't have to investigate those things further. I know I'm not gonna call "new" in the middle of a thing because I never do that. If you develop a programming style that you find is effective, simply leaning on that style in your

5. **Allen** is an entrepreneur and engineer interested in working on the tools that drive digital creation. He is currently working on "4coder", a programming environment targeted at the problems of real-world high-end C/C++ problems, under the Handmade Network.
6. **Overloading** an operator replaces the functionality of that operator for a given class. For example, you could overload the assignment operator + on a list object to instead add the given other object to said list object.

own code will allow you to keep it flexible and easy to remember.

I think those are the two aspects. One of them is about becoming a good programmer, which is making sure that your style actually pays real benefits and not hypothetical unproven things. The other one is keeping the code small and straightforward. I think people overstate the difficulty of *Handmade Hero*; it's not that hard when the code is that small. If I tried doing that on the Unreal Engine codebase, I would not be coming back to it and know exactly where I left off. I'd be completely lost, and have to step through it in the debugger for six hours before I really knew.

Everyone's Own Coding Language

In a similar way, working with different peoples' coding styles is probably one of the biggest problems in programming today. If you imagine that programming is a two-part process, there are two things that are going on at any time. One is coming up with a "language", and the other is speaking in that "language".

In the industry, we talk about having a programming language, but we really don't have one that we use. What we have when we talk about a programming language is the building blocks for the actual language that we will use. It should probably be called a "programming alphabet" or "programming phonemes" if we're honest. Because what happens in a game engine is that first you take the language that you have, like C++, and you build your own "language" on top of it. This will be a sort of functional language of core things that everyone will use. It'll be used to talk about things like how the memory is managed and how to implement the render pipeline and how do we pass things back and forth, what is the job control[7] story, and so on.

Those specifics form a secondary language. It's a very recursive process. You could think of it as making lots of languages on top of each other. Some programming languages and projects

7. **Job control** is the control of multiple tasks on a computer system that may be "in-flight" at the same time. It requires proper allocation of resources and locked access to prevent deadlocks and failures.

are so poorly thought through that they end up with languages on top of languages on top of languages. If you think of each programmer as having their own ideal language they would like to see, that's the substrate that's right on top of the regular language that they all had to conform to. They're all fluent in that, presumably their own one. So when you bring two programmers together you essentially have a problem where they need to write a book together but one speaks French and the other speaks Spanish. They have to figure out how to come together to write the book, which would have to be in a Latin style typography but not actually in either of their languages.

So I think that the jury is still out. I might even say that the appellees and the plaintiffs and defendants are still out on what the right way to do that is. There's certainly procrustean approaches where companies will demand everyone have this many tabs and every class looks like this and there's a file for every block. It's like everyone must conform. There are other places that use a more laissez-faire approach, and they hope that everyone will figure out what to do at the boundaries.

I think that we don't know the answers to what's right about that and I would be absolutely lying if I said I thought I had a good solution to it.

Fluency, Efficiency, and Cooperation

There is no question in my mind that there is a loss of speed, efficacy and quality of code that comes from shifting from your native language to another one. Shifting from your native programming style into somebody else's costs you. You have to balance that cost against the fact that if we have more competent programmers on a project, assuming each of them is capable of writing something useful for the project, we can get more done.

While working on *The Witness*, I took a pretty massive productivity hit working in that codebase compared to my own. There's nothing you can do about that. I did build some of my own language in there. I put some of my own tools in there over time so that I can be effective, but it was on an on-demand basis. There's an entire article I wrote about a time when I had

a bug, which only came up because I made a wrong assumption about the math library. And that's the reason that I and Jon Blow are both able to write certain types of code quickly — because we both make assumptions about what our math libraries do. If we couldn't make those, we'd literally have to read the code for cross-product[8] or something, every time, to figure out whether it was right-handed or left-handed. The process would be much slower and there'd be way more bugs.

It's really hard to overstate just how important that is. I think a lot of people don't necessarily realize it, because when you program your first engine, you're usually not a very mature programmer. If you've never written an engine alone after you've had a lot of experience and are more self-confident in your skills, you may not realize how fast you actually are when everything is done the way you expect, because you've only ever done it at Valve or something where everything works maybe a little bit differently than how you would want it to. And so it's a very important thing to be aware of.

On that front, I think working with an existing engine would be really good for engine programmers to start with. In an engine that's made a bunch of decisions, whether they're right or wrong, you have to live with them. Because that's probably going to be your job when you're first starting out. For example, the Unreal Engine does different things wrong but you can't change those because they're baked in the architecture. So it would be important for new engine programmers to learn how you make these improvements, or how you make this one part better.

Handmade Hero as a Learning Tool

The goal of *Handmade Hero* was to stream it, because I wanted to have that complete record. I thought there were things people would learn by watching an engine programmer just do what

8. **Cross-product** is the 3D math operation where the input is two vectors and the output is one vector that's perpendicular to both input vectors. However, the direction of the output vector depends on whether the space is defined as left handed or right handed.

they do, rather than just telling them the result. I think it's been really good at that because I've had many people actually tell me that that was a big breakthrough for them. They would say, "Seeing what you actually *do* totally makes me feel more comfortable about doing this." So that, I think, was the main thing I was trying to do with *Handmade Hero*.

That's what I hope people get out of it. Everything else I have no idea, because the goal wasn't to teach people, say, 3D math. I just do that because I wanted people to see an explanation of those things as we used them. I don't know if it's a good way to learn it and I don't know what other things would be valuable. If I was gonna make a course for people to learn 3D math for engines, it wouldn't look like *Handmade Hero*. So there's definitely a lot of room for more learning materials, regardless of whether or not such resources are available right now.

So I don't think that *Handmade Hero* is the best we can do for learning specific, individual topics. I can say that for sure, because if I sat down to make something like that myself, it wouldn't look like *Handmade Hero*.

I don't think you ever want to use any one programmer's ideas or code as your sole resource. The reason for that is because everyone's brain works a little differently. That means the most efficient way to think through a problem is not the same for everybody. Unless your brain literally works exactly the same way that mine does (and there may be some people out there whose do), then the chances that my streams are telling you literally all the things you might want to know about programming is zero. There are going to be very valuable approaches and things that I might not be aware of that you will find effective. So I think it's important for a programmer's brain to be able to distinguish between those styles in a concrete way. That's the way you can find the primary styles that work for you.

As I got better at programming, I found myself actually thinking, "How do I figure out what elements of programming style are good and bad for me?" I don't care what people write in these books. I've realized that they don't really know what

they're talking about a lot of the time. I want to come up with ways I can evaluate this stuff for real, and I want to see how it works when I do it. I felt like I became a much better programmer when I realized there was a really big divide between the prevailing theories and actual programming practice. I think it's important for programmers to try to cross that divide for themselves. They're not going to cross it by watching a series like mine and assuming that everything I do is right. Even if everything I did *was* right for them, they still wouldn't have developed the critical faculties necessary to know why. I also suspect that for most programmers, not everything I do in the exact way I do it will be the way that they should do it, because they will go on to find different techniques that are better for them.

It's the Process, not the Product

If I had to start from square one on *Handmade Hero*, obviously there are things I would do differently, but that's because the point of *Handmade Hero* and for recording the process was that I wanted people to see me go down a few avenues before picking the one I wanted. I wanted them to see how an amorphous, unstructured piece of code resolves into a structured one.

For example, one thing we did over the past three weeks was we made the renderer into a standalone DLL[9]. Now anyone can use it to make a renderer that does all the things that *Handmade Hero*'s does. So the depth peeling[10], sprites, and the camera code… all that stuff is in a separate library now. So if you wanted to make a game that has an instant 3D-hybrid sprite block renderer, you just have one.

Those are exactly what I wanted people to see in *Handmade Hero*. So if I lost the whole thing, I almost would say I would start a different project. If I was going into *Handmade Hero* with the knowledge of how everything worked out, it would totally

9. A **DLL** is a dynamic-linked library which is Microsoft shared library concept which can be transported around easier than a project and contains information about the compiled project.
10. **Depth peeling** is a method of order-independent transparency when rendering 3D geometry. It determines what should be drawn on top by rendering multiple passes of a scene and comparing depths.

ruin the entire point of the series. Watching me type in code I already know how to do isn't worthless, but that's what a blog or GitHub repo is for. On *Handmade Hero*, it was critical that I not have a solution in mind when I started.

Reusing Past Subsystems

Utilizing code from past projects is not something I'm an expert on. I'm only 42 years old! Maybe when I'm 82 I'll be able to tell you if I had a piece of code that was worth reusing. There are a couple of things I can say along those lines. For the very first time, on *1935*[11], which is the main project I'm working on now, I decided that I never want to write a platform layer or standard library again. I'm expecting honestly this project is probably a five-year project, which is a long time for one programmer. It wouldn't be that much for a team if you think about it in programmer years, but for one person it's a lot.

So on this project, I felt like I was ready to do that for the first time. Whether it succeeds or not, I'm not sure. But what I will say is I took an approach that was slightly different from approaches I had taken before. I documented the whole development process. I have a notes file, and every time there is a question about what I'm going to do in the codebase, I document exactly *why* I thought there was a question. I discuss the solutions that I've tried. Finally, I explain why I ended up selecting the solution I did out of the ones that I tried. So there is literally, for the first time on any product that I've ever done, a 100% complete documentation of why every last thing works the way it does. Before this project, I never sat down and committed to thinking all these things through, because a lot of times to do a platform layer you just do it.

Justifying Brought-In Systems

Even for myself — who advocates a very limber style of programming where I don't think you do a lot of upfront design — even I want to spend more time recording and justifying decisions. I don't do them from the top down like a UML dia-

11. ***1935*** is an upcoming game from the MollyRocket team, which includes Casey.

gram[12] disaster situation or anything like that, but I do feel that a higher level of rigor is necessary. What I try to do in this situation is sort of play the devil's advocate; as if I was more than one programmer. I try to come at it from different angles. Can I justify it from a speed perspective? An ease-of-use perspective? A compatibility perspective? So we'll see if it works. Ask me in ten years!

Builds: Keep 'em Simple

In our current weird programming culture, there's a term called "build engineer"[13]. All I can say about that is the way to approach a build is to realize that computers today are so fast that almost any game engine you care to make, especially on a team as small as yours, can be built with a batch file in like ten seconds. Anything you do more complicated than that is a waste of your time. People throw in integrated continuous build servers and CMake[14] and Ninja[15] and distributed builds and Python, but none of it is necessary. All you need to do is just make a thing that says "compiler, here are the files, this is the executable I want, build." That's it!

On *Handmade Hero*, we showed how to do this on the first day. You don't need anything more complicated than that and I would encourage you to start there. Eventually, somewhere down the line, there are some arguments to be made for doing some of the slightly more complicated things, like a continuous integration server when you have multiple platforms. This is just so that not everyone has to have every devkit to make sure they don't break the build. Otherwise, a single-line build is the build you want. It shouldn't be more complicated than that.

Interview conducted September 8, 2018.

12. **Unified Modeling Language (UML)**, whose purpose is visually representing a software system with its actors and roles so that a programmer can better understand and design said system. Sometimes, UML diagrams can end up as a "disaster situation".
13. A **build engineer** is in charge of the infrastructure that builds a software application, as well as testing and troubleshooting code for before the software's release.
14. **CMake** is a cross-platform, open-source application for managing the build process of software in a compiler-independent way.
15. **Ninja** is a small build system that is designed to run builds as fast as possible.

An Engine Developer's Toolbox

Jeff Preshing

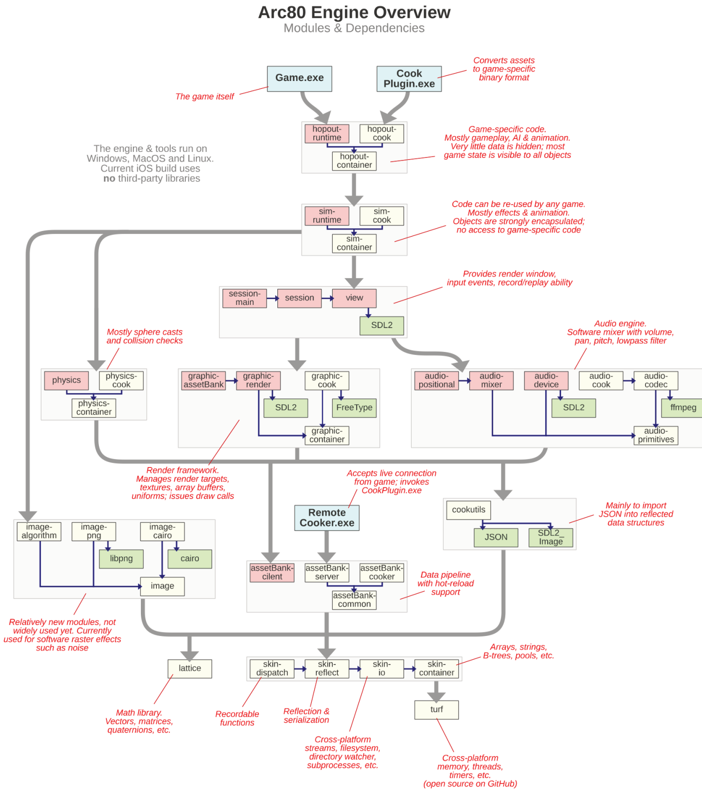
Jeff Preshing is a programmer with close to 20 years of experience working for various game companies and non-game companies. He recently wrote a C++ game engine from scratch and is using it to make a dope cartoon-action game for mobile. You should follow his blog at preshing.com. He likes rock climbing and Vindaloo curry.

The Arc80 Engine Architecture

Back when I was starting my career, I considered myself a good programmer, but I would look at existing games and game engines and just be unable to even fathom how everything all fits together. That's the most confusing part, and to illustrate, I've provided my own engine's architecture here. This can give you an architectural overview of my own game engine; it's something that I would have liked to see early on in my career. You can find other overviews like that online, but having a particular view of the modules, the dependencies[1] between them, and how an engine goes from high level to low level is a good

1. **Dependencies** are links that are required between programs, such that one program is reliant on another.

start. That was by far the most confusing thing: Getting the big picture of it all.



Arc80 Engine Overview

Engine Design Principles

I have my own set of personal design principles behind how I write and maintain my own engine. They probably won't apply to anybody else, but that's what I've chosen and I think it's an interesting way of looking at things.

To the extent that I can, I avoid building custom tools. I don't want to spend time building them. That's a big part of real AAA game development at a studio—because you've got hundreds of people on the team, the investment that you make in user-friendly tools pays off. That said, it's a big investment, and for me personally, I don't want to spend time on custom tools. To get around that, I leverage Blender[2]; I do all my level editing and character design in Blender.

Another one of my personal design principles is that I always want to focus on the application running on the device. All my effort is targeted at keeping that lean and mean and efficient, so rather than focus on tools, I focus on the game application itself.

The other principle that I follow to an extreme is maintaining reusable modules, because everyone likes modularity. The way I make sure that my modules are reusable is by building small applications that compile and link with individual components, and I can't do that unless the components are decoupled from each other. When I say components, every small box on my architecture diagram is a component. Each one has its own directory of header files, and when each box is compiled, it's a library and it has access to the headers of specific other modules that it's allowed to depend on. Because of that, I'm really aggressive in terms of modularity. What I found at game studios is that engines tend to be monolithic code bases, so I'm deliberately making an effort to go the other way. I would almost say that what I've got is more of an SDK[3] than an engine—I just happen to be making a game out of this SDK.

2. **Blender** is an open-source 3D computer graphics software used in creating 3D models, animations, and interactive applications.
3. A **software developer kit (SDK)** is a set of programs used in developing another program.

Creating Your First Engine

For my own engine, I write everything. I've written my own containers, string class, and file system class, which includes the physics and the audio engine. I use almost nothing from the standard library, except for type traits[4]. If you're a novice game engine programmer and are setting out to do your first project, *you don't want to do that*. For everything that I've written, there are really good open-source equivalents.

So I don't know if it's a good or bad idea, but one way to build your own initial game engine is to look at it as an integration exercise: Go get GLM[5], download Bullet[6], download some renderer like Horde3D, and so on. You've got libraries for a lot of things, like libraries that import 3D models and everything, so you can stitch together an engine out of those. I think that's a good way of going about it, since it will expose you to the interfaces to each component. That's the best starting point, and then depending on how your needs evolve, maybe you'll end up opening those black boxes and modifying things under the hood. For a beginner, you can integrate available code rather than doing everything from scratch, because that's taken me a long time. But that's just my style.

Skills of Engine Development

These were skills I didn't expect to need in the beginning of my career, but I ended up developing them during my career. The reality is that every engine programmer ends up exposed to them, so don't be surprised to find yourself getting better at these skills by necessity.

Systems Integration

I call the first one **Systems Integration**. It's actually a big cate-

4. **Type traits** define a compile-time templated-base interface to query or modify the properties of types.
5. **OpenGL Mathematics (GLM)** is a mathematics library based off of OpenGL specifications, that contains definitions for typical math constructs used in graphics.
6. **Bullet Physics** is a real-time physics simulation library, a physics engine, which simulate collisions for soft and rigid body dynamics.

gory, as it has to do with integrating third-party libraries—taking separate codebases and making them build and link together to run as one thing. Another systems integration exercise would be porting to new platforms, as well as doing big three-way merges (you've branched a library, integrated it at some point, then maybe you've made changes both to your code and inside the library itself). That happens a lot in the industry; months later, there's a new version of a library you want to integrate, which is the latest and greatest of that dependency. That can end up being a huge task that can take weeks or months to accomplish.

Between all of those things, Systems Integration is a valuable category of skills to have. However, a lot of C++ programmers want to avoid it because they hate it. Many C++ programmers hate integrating libraries; they want everything to be a header-only library, just to make their lives easier. If you learn how it works, though, you're doing yourself a big service. Getting more familiar with it and tools like CMake[7] to help you generate build pipelines instead of tweaking Visual Studio properties is just a good skill to have. It's especially useful when you're going cross-platform.

Low-level Debugging

Low-level Debugging is another great skill to have, especially when you are tracking down crashes in the final optimized build. Especially at the end of the project, you'll have testers hammering on the optimized build, so there's bound to be crashes and issues that don't happen in a debug build. The ability to diagnose and fix those bugs is useful because you want to ship a stable game! How you develop that skill is a different story, but at some point, you or someone on your team will have to go into the disassembly[8] window or the memory view and figure things out at that level. It's kind of considered a black art, which is a bit of a shame because it doesn't really have to be.

7. **CMake** is a cross-platform, open-source application for managing the build process of software in a compiler-independent way.
8. **Disassembly** is the assembly language, translated from machine code, of a program; it is the compiler's version of the program.

I started developing my low-level debugging skills when I had no other choice but to figure it out. In 2005, we needed to ship a project but it kept crashing. I knew it was possible to solve the problem because, at an earlier job, I had a teammate who would constantly go look at crash dumps and disassembly and memory views. I would always think, "What the heck is this guy doing?" I asked him how he did it, but he never explained it to me; he just said: "I like computers." So I knew that was a skill that was possible to have. I don't remember exactly how I picked it up, but at some point, that became my bread and butter at Ubisoft. My teammates there would always come to me to if they had a crash and I would help them out. Eventually, it got to the point where Ubisoft started giving an in-house course on this subject. In my experience, it's something that's passed on from one person to the other, and yet no one really seems to want to get down to that level. Everyone's more interested in what sort of high-level programming paradigms can help us be more productive, but the most productive thing is to ship the game, and to ship the game you have to fix these issues at a low level.

Profiling

The next skill on the list is **Profiling.** Everyone agrees that it's important—you want to profile before you decide where and how you should optimize. How you use profilers is kind of similar to low-level debugging in that it's a skill passed on from one person to another on a project. I would point to a talk I gave on profiling at a student conference. I think it's a pretty good introduction to the different types of profilers and examples of how you would use them in practice.

Concurrency

The fourth skill worth developing is **Concurrency**, or multi-threading, which is probably no surprise considering everyone talks about it. Everyone knows it's part of life as a game developer.

Iterative Development

Number five, **Iterative Development,** is the ability to look at feature goals and envision how it'll work at a level. If you can break that into bite-sized tasks and decide what's your rollout plan (which tasks you'll do in what order) is a good skill. Managers love it because they want these enumerated tasks, which helps them create a schedule.

Development Journal

I also think an engine developer should keep a **Journal**. When you're working on a task or debugging a problem, you might open NotePad or a text editor and start taking notes, you could copy and paste call stacks[9], and maybe you take notes when you're researching things on Google and Stack Overflow or even while brainstorming. I especially make a lot of notes when I'm brainstorming—looking at all my options and alternatives and trying to figure out what's the best one.

So when I talk about keeping a journal, I'm not talking about keeping a diary of what I did each day. What I mean is having a place to store your notes so you can go back to them in the future. The way I personally do this is I have a folder full of text files, and it's just organized by date. I don't even try to organize by category. I just have one text file per month and at the start of the day, if I have something to note, I put a heading for the date and I jot things down. I can't work without doing that personally, because I'm constantly trying to remember what my observations were on a subject in the past. It's great to be able to search that directory of my notes and refresh my memory that way. I think it's a useful skill, but maybe every person has their own way of working in that respect.

Reflection and Serialization

The next two things on my list of skills are **Reflection and Serialization**. In the industry, every game engine has their own

9. The **call stack** is a stack data structure that stores the information about the active routines of a computer. It can be walked up or down into the callee functions and function definitions, respectively.

approach for this. You don't even need to have an approach at all, but just be aware that there are varying approaches to reflecting data structures. There are different reasons to have reflection, ranging from serialization to managing shader parameters to networking. Being able to think in those terms is a particularly useful skill for game development because C++ has nothing for runtime reflection. Well, there is RTTI[10], but RTTI doesn't provide any information about data members, so it can't help you implement things like serialization.

Professionalism

The final thing is a soft skill: being **Professional**. You'll have to do most of your work on a team, so obviously it's a matter of treating your teammates with respect. This was a mistake I made at the beginning, when I really wanted to advance in my career and I really wanted to shine as a good contributor to the project. A better way of thinking, though, is to help make sure that the game ships. Five to ten years from now, no one will remember who shined and who didn't, but if you have a shipped project on your resume, that's like currency; employers love to see that. So you need to stay professional and not get attached to your way of doing things. Letting other people tackle the problem their way, even if you don't think it's the best way, will minimize friction on the team, and that helps the game ship.

Console Development Experiences

The first time I did a seek-free loading[11] system, it was only for the Xbox console on the game *Rainbow Six: Lockdown*. Since we were loading from DVDs, seeks[12] were expensive, so you wanted to avoid them. My job was looking at the problem of how the game is opening all these files and reading from different places.

10. **Run-time type information/identification (RTTI)** is specific to C++ in that information about an object's type is available at runtime.
11. **Seek-free loading** is a system which is able to read a file "free" (without) "seeking", (searching) a file, for data/position within a file. Seeking causes disk activity which is generally slower than CPU performance.
12. A **seek** is a programming concept related to file reading, where a program has a file pointer associated with a position and a seek moves the pointer to a specific position within that file.

All I did was implement a mode where the game runs, and as it starts up it's logging which files it's reading from and in what positions. It then takes that data and makes a linear version of the same data with no seeks. I also added another mode of running the game that uses the seek-free files, so I was able to just focus on the strategy of how and when the data is accessed. That also included fallbacks, so if the game loaded a little differently the next time, it wouldn't totally break. At that time, though, I didn't have to worry so much about platform differences. The second time that I did it I already had experience with the strategy, but I had to do it in a cross-platform way: for PC, PS3, and Xbox 360.

That meant it was just a matter of reimplementing the same thing, but using cross-platform primitives. The engine already had some cross-platform wrappers for low-level things like opening files and such, but not everything. I remember on the PS3, there was a limit to how many files you could have open, and that limit was very small. At some point, I had to make a wrapper layer between the engine and the hardware that let the game pretend it was opening a lot of files, but if it wasn't actually reading from those open file handles—I was closing the files and it would be opening them on next access.

The reason I had to do a lot of PS3 optimization was that the *Rainbow Six: Vegas* engine was actually a branch of Unreal Engine 3. That project actually branched very early, before Unreal 3 was even officialized, so we didn't have a lot of the work that Epic Games did that we could just integrate. That was actually one of my favorite projects because when *Rainbow Six: Vegas* shipped on PlayStation 3, it didn't perform very well: It was pixely, it was blurry, and it had a low framerate. But on *Rainbow Six: Vegas 2,* we optimized things and made it look good. The PS3 version almost became indistinguishable from the Xbox 360 version. We didn't even use the SPU's[13] a whole lot, as I recall. I remember SPU's were used by audio mixing, but I don't recall that we used them for the engine itself. Most of it was basics, like

13. A **Synergistic Processing Unit (SPU)** is the processing unit associated with the Cell architecture of the PS3. The PS3 had seven as part of its hardware, only six of which were usable by game developers.

making sure you had the best frame buffer[14] pixel format for the GPU because the PS3 needed a specific set of conditions maintained for Z-buffering[15] to work in the optimal way. It was hierarchical Z, and it was very easy to break that, so using the tools that Sony gave us to identify where it was broken and fixing those, that was a huge boost. Avoiding redundant state changes being sent to the GPU was also a huge win. So a lot of it was basics, not crazy concurrent SPU stuff. I saw a lot more SPU stuff on later projects, but I didn't gain that much experience developing for SPU's myself. I did a lot of debugging on them for sure, but not so much on the development side.

How Profiling Can Differ

Profiling on *Child of Light* was completely different from profiling on *Assassin's Creed: Unity*. *Child of Light* was a much simpler game engine by far; when you put the controller down and let the character sit there in an environment, activity on the CPU and GPU was pretty consistent from one frame to the next. With *Assassin's Creed: Unity*, on the other hand, it was a totally different thing because the assassin was often in a crowd, and the crowd is walking all over the place and bumping into each other. That movement causes a lot of fluctuation in CPU activity from one frame to the next, and that made it harder to profile. It kind of bummed me out a little bit, but it seemed like most of the *Assassin's Creed* team's approach to optimization was just moving stuff around from one core to the next. They have this job system and you capture a profile and you see which jobs ended up on which cores, and if there were holes in the schedule, they would try to move jobs around. There was not as much direct optimization going on. To be fair, it was such a mature engine that it was hard to find those optimizations.

14. **Frame buffers** are a portion of RAM containing a bitmap of the display, containing the data for that given frame on the video display.
15. The **z-buffer** also known as the depth buffer, contains information regarding the distance from the camera, the depth. **Z-buffering** can also refer to the technique in which pixels are culled from the frame, not rendered, because another pixel's depth is closer to the camera, therefore the pixel in the background is being covered.

The Game Engine Marketplace

If you ask me, "What's the next big thing in game engines?" I would say the Arc80 Engine, obviously! Just kidding. I mean, I would love it if I could use the Arc80 Engine for the rest of my career. But I don't believe in the "next big thing" in terms of a predetermined path that the industry is inevitably walking along. Right now, there are people who think that real-time ray tracing will be the next big thing in game engines, and there are people who think that cloud gaming is the future. I don't know about either of those things. I do know that history is littered with inventions that people thought were going to become huge, but weren't, because market forces just didn't support the idea.

Game engines are just another marketplace. The ideas that succeed in this market will be the ones that meet a demand, either by enabling more interesting games for consumers, by making developers more productive, or both, or in a different way altogether. The only way something will become "the next big thing" is if an idea is so good, so compelling that a big segment of the industry adopts it. But by necessity, that will always be hard to predict, because if an idea was both compelling and obvious, people would already have adopted it! The point is that the future of game engines has not yet been written. As engine programmers, we're the ones who get to write it.

Interview conducted October 24, 2018.

Engine Programming is All Plumbing

Amandine Coget

***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.*

About Amandine

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. 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], and have been freelancing since then. As for my hobbies,

1. **BitSquid**, more modernly known as Autodesk Stingray, is a discontinued game engine from Stockholm, Sweden. End of sale was announced for January 7,

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 glass-blowing 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.

2018, and afterward it became a plugin for Autodesk 3DS Max known as 3DS Max Interactive.

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's take on IMGUI[2] (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[3], which is heavily retained[4], 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

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)
3. **Scaleform** is a vector graphics rendering engine used to display Adobe Flash-based user interfaces and HUDs for video games.
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.

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] 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] 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

5. **Frostbite** is EA's proprietary game engine used across most of their studios.
6. **HUD** stands for Heads-Up-Display. It usually refers to overlay on the screen that presents important information to the player.

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 an 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], 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]? 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

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.
8. **Retro compatibility**, also known as backward compatibility, is when a system is setup such that it works with legacy code/input.

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], 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]?

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.
10. **std::iterator** is a C++ type that can be used to iterate through collections of elements based on that collection.

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. While for a 2D game, 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 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? 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.

The Definition and Beginning of a Game Engine

Adam Serdar

__Adam Serdar__ is a Senior Game Engineer at Schell Games where he works on projects that need server technology, graphic effects, or frame rate optimization.

Most recently, Serdar was part of the team that created HoloLAB Champions*, a virtual reality lab practice game in which he was deeply involved with its integral game systems, including how virtual objects could realistically simulate solids and liquids and how they are handled. When he's not coding at Schell Games, where he has been employed for twelve years, he is working on costume and robotics development projects as well as learning, practicing, and teaching Kung Fu.*

A Game Engine and its Needs

A game engine is, in its broadest sense, a set of tools that enables game creation. Depending on your platform, this could be as simple as having a scripting language which makes it easier for developers to interact and develop. Oftentimes an engine is a collection of tools, an update loop, and a scripting language. The Unity game engine accesses the update loop through the

MonoBehaviour[1] system, as well as tool development with ScriptableObjects[2]. An engine can also be a simple system which has an update loop with rendering, giving the developer the ability to display bitmaps on screen. The simple engine can be expanded to have an animation system or other systems that the specific game being developed requires. That engine will grow and become more comprehensive as more developers spend time with it.

Most engines are going to have some sort of graphics, some sort of at least minimal physics, some sort of audio— mainly because those are going to be your most common systems that every game is going to want. Graphics are going to be one of the most common things that your game engine will do. Graphics of an engine can be simplified to the type of asset that needs to be imported because the graphics engine only needs to support these types of files. A simpler 2D game may only need to be able to import bitmaps, but if you're developing a 3D game models with animation might need to be supported. However, I know of at least three games off-hand that are completely audio-based. They wouldn't have any need for graphics. They're just showing a blank screen. They are more focused on audio bouncing off physical representations of the environment, kind of like a sonar-type game. If you're building an engine that's all about simulating that kind of system, the engine doesn't need to support graphics. Scripting languages are extremely nice to have so that programming is simpler and less brain-taxing; however, it's not required and you can usually just write code in the exact same language as the engine.

Engine vs. Framework: What's the Difference?

If you think of the line between an engine and a framework as layers, the lowest level is certainly the engine. As you work more with that engine, you might find yourself developing a frame-

1. **MonoBehaviour** is the base class within Unity that all components which attach to GameObjects must derive from, it has methods for start, update, and destroy.
2. **ScriptableObjects**, are scripts which cannot be attached to GameObjects but still store (serialize) user data.

work, and the question of whether that's part of the engine or not depends on the number of users the framework has relative to the engine. The framework developed at Schell Games, which is built on top of Unity, is exclusive to Schell Games and the employees who take it home. That being said, sometimes a framework can get folded into the engine. For our case at Schell Games, that would require our framework to be assimilated by Unity. Sometimes features in frameworks are built prior to being included in the engine, and sometimes the specific implementation of the framework is better suited for our development. *A framework is additive: It's a layer between the final game code, which is very specific to a game, and the engine.* The line between the two is fuzzy; however, when there is game-specific code shared between multiple projects it belongs in a framework-type system. And although an engine is constantly being updated, it is relatively constant and self-contained.

The Engine Dictates the Game

As you're building a project, from a design perspective, you look at the nails and hammers you have and you say, "this, this, and this— those are great, we just need to add this extra nail." And sometimes this extra nail is system-level code. Or it's game-specific code.

Having access to the entirety of your engine code does allow you to take a feature you need and scope out what's required of the team. And in picking your game, you get a pretty good idea of the features, so you know at least nominally what features you want in your engine. In theory, your system should be fine.

With our projects, we often have a little bit of that decision-making for which engine we are going to use. Some of that comes from the client; some of that comes from our internal experience. And we really have to weigh the fact that if we don't use Unity, or we don't use some other familiar engine, there's a certain amount of cost with ramping up to that other system. We'd either have to tack that on as a cost or take that as earning more experience with this new system.

In the Isetta Engine's[3] case, the team has a twin-stick shooter game that they want to make, so that tells them which features to include in the engine. They themselves are gonna have to decide whether they have enough time to implement all of those features and make it as close as possible to this first game.

This is not the way of things usually. In fact, I'd go so far as to say most of the time, you already have some level of engine already built, so it's usually a little closer to taking a design proposal and determining how to use our systems to make that. For example, a game by EA (*Madden*, for instance), they typically just use the features already existing in their engine while upgrading them. If they need to move the franchise to a brand new platform, then they would have to upgrade the engine for that.

Adding New Engine Features

Updating an existing engine is somewhat iterative. If we're looking at *Madden* specifically, the very basic rules are always going to be the same, but there are always going to be features that the team would like to add. Offhand, what I'm thinking is the first iteration on the PlayStation 2. The team needed to render players, and the players also had to be customizable enough to be recognizable. Different sizes, widths, girths— lots of different qualities. And the different players need to interact reasonably well.

In the second iteration, the team considers a more advanced problem. What if three or four players all collide into the center? Initially, the implementation may require players to collide in a sequence, like the first two, then the next two, etc. Is that good enough? The team may decide that they want some animation-blended system[4] where all of the players come into a giant huddle to make the pile-ups even more realistic.

3. The **Isetta Engine** is a game engine developed by the editors of this book for the sake of learning how to develop a game engine. See isetta.io for more information.
4. An **animation-blended system** can be a graph of multiple animations and transitions from an animation to another, i.e., an idle animation to a walk animation, and the blend system is how the animations are "mixed" together. It extrapolates from the starting animation to the ending animation.

Also, as you're moving from Gen 1 PlayStation 4 to Gen N PlayStation 4, there's a lot of performance gains that can be made. Maybe a core[5] on the PlayStation 4 was hardly being used before, so the team decides to make it the "audience core." Instead of little stick figures on flats, the audience is now full of 3D instanced people who are animated!

There are bigger leaps, too. For example, moving a system from PlayStation 3 to PlayStation 4, or responding to years of stagnation and infrequent updates. Where that's driven from is an interesting question. Oftentimes, it's both the engine guys wanting to do something cool that they couldn't before and the designers or scripters wanting to fix the weird things they've had to deal with.

Iterative vs Waterfall for Engine Design

For the design and development of a project, it is better to have an iterative approach to that of a waterfall schedule[6]. Start simple then layer in more features, with the caveat of removing or adjusting particular sections when they don't make sense. I recommend an iterative approach to projects, unless time becomes an issue. Oftentimes, with shorter projects, prototypes turn into the final product quicker than you'd expect, or at least large chunks of code are copied regardless of the quality. Ideally, you can do iteration, but practically it just may not happen.

Neglected Systems

From my perspective, usually graphics is the first thing people are worrying about, and poor audio is always last. That tends to be my experience in a development space of working with N number of features. We'll get to audio when we get to audio, but the shiny and pretty stuff tends to come first.

Is anything neglected? I would say no. Even audio has a bunch

5. A **core** refers to a CPU in a multi-core processor; it is one of the processing units in the single computing component that read and execute machine instructions.
6. A **waterfall schedule** is a linear schedule where each subsequent item is dependent on the previous components being completed. It is less iterative and flexible because the flow is usually mono-directional.

of people that are very enthusiastic about the systems that they care about, and even Unity is spending time creating brand new— well, more likely importing new systems to make their audio better. Usually, if it's neglected from a Unity standpoint, it means Unity is at least marginally aware of it and is probably looking for a solution to integrate cleanly. Some of their newest systems are mostly graphics, like the node-based shader system[7], the Scriptable Render Pipeline[8], that sort of thing. They're going through a very radical shift over what you can and can't do with the graphics pipeline, which I'm personally excited about.

Industry Standards for Engine Systems

There's no huge standard for engine systems other than the APIs that other systems are using. Usually, if you want to make your own custom audio system, then you're going to have to spend a lot of time building up threads, make sure they play through the systems appropriately, how they get loaded, and all that. Or you can spend time learning the APIs for OpenAL[9]or whatever it is, and then they kind of have a way they're expected to be used.

It's kind of about what you expect to write yourself and what you expect to use as an external library. DirectX[10] and OpenGL[11] will have very specific calls that you're basically

7. **Node-based** means the interface is visual with components, "boxes", that are connected to each other with outputs connected to inputs. A **shader** is a program that alters the graphical look of an object. A **node-based shader** system means the shaders are edited through nodes.
8. The **Scriptable Render Pipeline** is a system in Unity that allows the game developer to configure and control the graphics and rendering process via high-level scripting.
9. **Open Audio Library**, or **OpenAL**, is an audio library used for games, although it contains the word open it actually isn't open-sourced. Its open-source counterpart is OpenALSoft.
10. Microsoft **DirectX** is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming, on Microsoft platforms, like Windows and Xbox. It is most known for Direct3D which is the graphics API used for creating windows and rendering, and serves similar purposes as OpenGL.
11. **OpenGL**, short for Open Graphics Library, is 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.

required to do in a proper order or it's just not gonna do what you want it to do! It's more about making sure your system flows with the interface of the systems that you're using externally.

It's been a while since I've looked at those low-level APIs, and (when I was in the position) I definitely didn't want to write most of it myself. While I wanted low-level access to OpenGL or DirectX so I could do fancy graphics stuff, I was less worried about physics or audio or whatever because I wouldn't have to rewrite or force it. Much like building most games, building a comprehensive engine these days is teamwork. And oftentimes, that's finding a thing you're interested in, and working on it 'til it's awesome. And that's probably true for game development as well. If you've got some guy that hates UI and you push him onto UI, and you're surprised when it doesn't work out so well—whose fault is that?

Adam's Engine-Building Experience

At the time of building my first engine, I was in Panda3D[12] and Unity, and I wanted to know more about that rendering pipeline and what was required. And so, the engine that I built was very simple but very dedicated to, having a mesh, shader, and fancy particles moving around just on the GPU with nothing else knowing about it. It was a bit more focused rather than a generalist system. But I also wasn't expecting to actually make a game with this without significant investment in it.

You can spend a lot of time building an engine and then building a game, and then realizing as soon as you finish the game, it's now five-year-old technology. It may be cool, but it couldn't keep up to date with teams of programmers and artists and all these guys working together to build... for instance, in a horror game, where you want very interesting graphical effects. If you make your own engine, you've got very precise control over what's being rendered! Or, a team of people could have made the same thing in Unity, for instance, and had it out the door years before you're

12. **Panda3D** is a game engine, a framework for 3D rendering and game development for Python and C++ programs. It was originally developed by Disney and expanded by past ETC projects.

finished with the engine. So it's really a trade-off thing from a professional point of view, and that's why I chose to do a more focused deep-dive of asking what I'd need if I was to get access to the core render loop[13] of Unity. And that's kind of what my stupid little engine explored.

Integrating Libraries into an Engine

Once you have a whiteboard plan, definitely set up a version control system. In determining which libraries could be useful it is better to consider the feature set your project needs to support. Obviously, if you're not doing a lot of string manipulation, don't look for a library that does that. When using a library for the first time, there is some time lost to setup and figuring how to properly configure your IDE to work with the library.

A simple decision you may have early on is determining whether you are supporting OpenGL or DirectX. Does being close to open source matter to your project? This decision will affect discussions with all libraries you maybe thinking of including. Another benefit of open source libraries is you can modify the code if you aren't happy with how the library is operating; you have control. Another factor to consider in deciding on libraries is looking for a particular set of specifications/requirements for it to satisfy your project's needs. If you are going to have physics, you'll probably want to use Nvidia's PhysX[14] system to have all the computation on the GPU. The audio will usually run on its own thread. I would lay out each system the engine will include.

How Existing Engines Limit Game Development

The way I like to talk about Unity is it's very good at making 95% of the game, and that last 5% is going to be like pulling teeth. Performance is an issue, though they're getting better about that.

13. **Core render loop** is the loop where the rendering function is called. The way the rendering occurs varies from engine to engine, but is usually performed at the end of the main game loop.
14. **NVIDIA PhysX** is a proprietary real-time physics engine SDK created by NVIDIA. It is used in most commercial game engines such as Unity, Unreal, and Lumberyard.

VR is currently kind of "hacked" into Unity, and it's almost good enough, and that's where I think the scriptable render pipeline is going to be good enough— or at least pretty good.

On the other hand, my understanding is that Unreal from an engine perspective is very difficult to modify. It's a huge codebase at this point and recompiling that beast is an undertaking. Unreal has a lot of good things going for it, including graphics fidelity. You'll hear the insult: "Oh, that looks like a Unity game." You don't hear the same insult of "Oh, that looks like an Unreal game," because Unreal looks pretty awesome! If you need to do tweaking to that, it then gets much more difficult to do. That's my understanding, though I have not used it in years.

The Relationships of Engines in the Industry

I think you're starting to see a lot of cross-pollination between Unreal and Unity and other systems. Unity was always very good at making something quick— real quick— and making it playable and fun. If they want it to look great, that means spending more time; that's the last 5% sometimes. Whereas Unreal tends to have all of those pretty features already active, and as long as you know the gameplay scripting and the Blueprints[15] and whatnot, you're going to have a pretty good time. If you need to constantly tweak what it's doing, though, you're going to have a different kind of time.

The funny thing is we are looking into possibly taking at least some of the rendering tech of Unreal and putting it into Unity. Then people tell us, "this isn't a Unity game, it looks like Unreal!" And we'll say, "yes, that's exactly what we're going for!" We'll see if that really pans out, though. There's a couple interesting rendering systems that they've got that the scriptable render pipeline might make it very feasible to automate the process. But, who knows, that's future-seeing.

Interview conducted May 15, 2018.

15. Unreal's **Blueprint Visual Scripting System** is the node-based scripting in the Unreal Engine used for gameplay scripting.

Growing Pains in Engine Development

Aras Pranckevičius

***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.*

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, like my recent path tracer[1] series of blogs[2], 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[3], 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 Render-Ware[4] 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[5]. These days, you just export any FBX[6] or glTF[7] file and

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.
2. https://aras-p.info/blog/2018/03/28/Daily-Pathtracer-Part-0-Intro/
3. The **demoscene** is an international computer art subculture focused on producing demos, which are self-contained, sometimes extremely small, audio-visual computer programs.
4. **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.
5. **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.
6. **FBX** is a proprietary file format owned by Autodesk that is mostly commonly used for 3D model and animation data within the games industry.
7. **GL Transmission Format (glTF)** is a royalty-free file format for 3D scenes and models using the JSON standard.

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[8], 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[9] for the UI and the various UNIX[10] assumptions

8. **Open Dynamics Engine (ODE)**, is a free and open source physics engine written in C/C++ that can do both rigid body dynamics simulation and collision detection.
9. **Cocoa** is Apple's native object-oriented API for macOS.
10. **UNIX** is a family of multitasking, multiuser operating systems that derive from the original AT&T Unix, originally developed at Ken Thompson, Dennis

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[11] only. Neither Metal[12] nor any other alternatives existed at the time, so OpenGL was our only option. From there, we started to add Direct3D[13] 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[14]— 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,

Ritchie, and others at Bell Labs. Its main comparable is Microsoft's DOS, which is mono-task and mono-user.

11. **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.
12. **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.
13. **Direct3D** is a graphics API within Microsoft DirectX used for creating windows and rendering, and serves similar purposes as OpenGL.
14. 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.

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 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[15] 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

15. **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)

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[16], 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 six 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[17]. 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

16. **Vulkan** is a low-overhead, cross-platform 3D graphics and compute API that targets high-performance real-time 3D graphics, such as video games.
17. A **package manager** is a system which handles the installing, updating, configuring, and removing of a collection of software libraries.

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[18], 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 (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.

18. A **socket** is an internal endpoint for sending or receiving data within a node on a computer network.

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 are about ten people who are on Windows but run everything from a Cygwin[19] 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. 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, 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

19. **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.

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 modules, but the risk is going overboard with modularity. For everything that's good about JavaScript npm[20], 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

20. **npm** is a package manager for the JavaScript programming language. It's the default package manager for the JavaScript runtime environment Node.js.

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.

Wisdom from Working at AAA Studios for 15 Years

Elan Ruskin

***Elan Ruskin** is a senior engine programmer at Insomniac Games, where he has worked on critically-acclaimed titles including* Marvel's Spider-Man *and* Ratchet & Clank. *Prior to his time at Insomniac, Elan worked at Valve, Naughty Dog, and Maxis on many of their flagship titles as a gameplay and engine programmer. When he's not programming, Elan enjoys theater, music, and* Star Trek.

Designer-Driven Tools

The real advantage of data-driven systems is that it's designer-driven; you're decreasing the iteration time for the designers. It puts the ability to make and see changes into the hands of the person actually making the content, and away from the programmer-compiler loop which requires programmers to develop and compile before any change can take place. The problems with data-driven design are 1) the code has to be a little more complex to support this flexibility and 2) you're loading bulkier content. You can get around that, though, if you use

a builder[1] to pack down the content into something that code can load more efficiently.

What surprised me about data-driven systems is that it ended up being much less of a problem than I expected. It turns out that you load things relatively infrequently, and if you do end up loading them frequently, you can bake them down. So as much as it would make the me of 20 years ago sad to hear me say it, having things in an open-text format that gets parsed turns out just not to be a performance issue. I'm not saying performance doesn't matter! But that turned out to not be the issue.

The other problem with data-driven design is that people will do weird, unexpected, and strange things with data that you didn't anticipate and will possibly break your systems. That's because the connection between changing the data and something in the game breaking is not quite as obvious as it is with code, where you can set a breakpoint and see exactly what happened. The importance of good error reporting, diagnostics, and designing the authoring tools in a way that prevents people from getting themselves into trouble was not clear to me when I began engine programming.

With tools, the things that you can't anticipate are usually the problem, because the designers and artists are always trying to solve their own problems. They're not out to break the tools or fumble around oafishly; they have specific needs, like "I need this tree to have another tree on top of it so I can turn one of them off and the other one on, because it's winter and we need the leaves to be gone." In this scenario, they might not know that if you have two things in the same place at the same time, it causes a problem. That's something they would have no way to know about until they did it. So really, the way to deal with that is to find a way to prevent people from making the mistake, which would then make the causal connection obvious. Ideally, we should make it impossible to do bad things, but again, that

1. A **builder** is a tool used to process assets from their editable forms (files editable by external software) into a more compact, unreadable file to be used by the engine for a game. The file format is typically proprietary and specific to the engine, and engine metadata is stored within the file.

comes back to anticipating things. It's also not a reliable strategy to just go over and yell at people for having done the content wrong or give them a gigantic document that explains how to use your tools. You can't expect people to hold that much content in their heads at once.

When developing tools for writers, an assumption a lot of people make is that writers are not technical and therefore need easier tools, which is completely untrue. Writers can learn computers as well as anyone else! What I learned from my time at Valve is writers need flexibility; any given line of dialogue goes through many iterations to get it right.

The thing to be cognizant of is that writers are part of a whole pipeline of content that has to get made. The writer's text appears in the game while it's still being prototyped, but even at that time we have to cast the voice actor and record them. Then, we put the voiced lines into the game only to realize the dialogue is clunky. So we have to change the line, go back to the booth, and repeat the process. This puts the writers in the middle of a pipeline that has audio waggling at the other end. The advantage of building a complete suite of tools is you can integrate the whole process of tracking where lines of dialogue are located in the game, as well as who's been cast to play it, and whether or not it's been recorded and localized yet. In Campo Santo's talk on *Firewatch*, they discussed how they integrated the dialogue system with the "recording-tracking" system. Taking that approach saved them a lot of agita.

Intimate Bond of Engine and Tools

At Insomniac, the engine team and the tools team are the same team. That works well for us because the team is not especially big, and because the engine and the tools are intimately bound. The engine is loading the assets, the builders are cooking the assets into a binary[2], and the tools are feeding the things to the builders. These are not separate operations; it's all the same

2. **Binary files** are files stored in binary format, a format that is easily computer-readable but not really human readable. These are more compact in size than human readable files.

lump of data. As teams grow, you'll need to specialize the labor because they are different skills to an extent. I personally don't think there's that much value in separating the engine runtime from the tools in terms of being different teams, unless your studio is gigantic. In that case, you have to for organizational purposes.

Along the same lines, it's almost a necessity to version the tools and the engines together. Part of this is the obvious reason that if you change the runtime format and the tools need to export in your format, the engine needs to be able to read it. Again, they're operating on the same data. What's more, anytime you need a new capability the engine, the tools have to support it, so they really move in lockstep.

One of our attempts at improving our tools ecosystem was to use web tools. For the entire rundown of why Insomniac went with web tools and why we stopped using them, you should see Andreas Frederiksson's GDC talk. The reason for moving towards web tools was that we thought it would be much more flexible to make a UI in the web, and also that it would be easier to hire people who have a web UI experience than people who have C++ UI experience.

That just turned out not to be the case. We ended up hiring the people that we would hire anyway, and then teaching them JavaScript. What's more, the scaling issues of web tools are enormous. The web is good at doing 100 or 200 of something; it's not so good at doing 30,000 of something. So just performance and memory were gigantic issues, in addition to all the other issues like Chrome continually breaking underneath their feeds, JavaScript is just bad!

On the usability side, we made our new web tools work almost exactly like the old tools, only with less bugs. We tried to keep the interface consistent. The problems that we ran into were that the new tools didn't have all of the features of the old tools to begin with, because we couldn't rebuild everything at once all in one piece. As a result, the team would have to learn how to work around the missing pieces. Because we kept the the work-

flow the same, it was fine, plus everything got faster and less buggy.

We have also made several attempts at making a tool to track feature regression. We wrote a tool that loads every level of the game, takes a snapshot of the memory, and then unloads it and repeats the process. From there it would just create this whole spreadsheet report that nobody ever looked at, because it was always days out of date. We had another tool that was meant to interrogate a level; it would look at all of its dependencies and the dependencies' dependencies, and recursively try to see how much memory everything would consume. The problem of that is it's an estimate, because you often don't know how much runtime memory something is going to consume until you've loaded it due to dependencies. People will write code that causes dependent allocations that aren't even in the asset. But trying to detect memory consumption outside of runtime is a bad approach. If you're building an engine now, try really, really hard so you can look at an asset on-disk and know how much memory it's going to take up. Doing that saves you so much agitation. Failing that, write a regression tester, go into the level, and see the footprints. Keeping regression testing working— keeping the whole machinery of it working— is a full-time job. At least in a AAA-size team, somebody has to keep that pipeline going. We just didn't have the manpower to maintain that machinery, and that's why it fell down.

Allocate, Allocate, Reallocate

Memory allocation is always a problem. I don't just mean that the actual use of memory is a problem— the whole apparatus for allocating memory is only becoming more difficult as costs get bigger and we have more memory to allocate. A few companies that have had fixed memory pools[3] have gone back to using dynamic memory pools[4], just because of how much stuff was coming in and out of memory. It was a nightmare to fix it all.

3. **Fixed memory pools** is a data structure for dynamic allocation of fixed block-size memory chunks.
4. **Dynamic memory pools** are pools in which memory sizes are determined during runtime, and are changing per allocation rather than being fixed for all.

I think people are trying to find a way to get back to more static allocations[5], or at least block allocations fixed in size. When I say that, I mean the size of the individual allocations being fixed as opposed to the whole pool being fixed ahead of time. It's just hard, and that's why no one is doing it. Maybe someone much smarter than me is doing it! At Insomniac, the way we handle memory allocation is we have one allocator[6] for assets, a different allocator for render memory texture and graphics, a different one for physics, and one more for gameplay components. Each pool has different characteristics, because we're trying to allocate things of different size with different lifetimes. We have different pool allocators for different purposes. Insomniac's allocators are actually not pooled, because the assets are all different sizes. That said, we do use pooled allocators for stuff like physical objects; for example, pedestrians in *Spider-Man* are coming out of the pool. So in that situation, they are used for things that are all of the same type.

Balancing memory usage is an ongoing, iterative process. As you're making your level and you realize you need a few more textures over here, you may have to take some geometric complexity out somewhere else. You might need to put in some more actors, so the textures have to go. It's this continual give-and-take of budgeting. The upshot is that you really need good budget reporting, even more so than being able to decide what your budget is ahead of time. When people put in content, they need to know the weight of the content and they need to know the "pie chart" of where all the memory is going. Otherwise, you have no way to make trade-offs.

Effectively Using Profiling

The ability to sum together scopes across an entire frame is important. That's because you can have a profile scope that says how long to tick this asset's physics component, so if you're

5. **Static allocations** is the allocation of memory at compile time, therefore faster than dynamic because the computer doesn't need to switch into kernel mode to grab more memory.
6. An **allocator** is a data structure that encapsulates memory management and doles out memory on request. There are different types of allocators based on the needs (amount and lifetime) of the memory.

doing a hierarchy then it's gonna appear a thousand times. It's helpful to aggregate that whole thing together and be able to plot the aggregated number. A convenient way of exporting a report from a run of the game is as a spreadsheet that you can then import into Excel. Because then, when I'm making a change, I can run the game before the change and then after the change as control and experimental groups. I can also do a Student T-test[7] between them; doing statistics becomes more important. Otherwise when you make a change, you don't know if you actually fixed anything. Also, presenting the data from the the profiling part of the profiler is not that hard; it's presenting the data to the people who can act on it that's hard.

For a discussion of how I tracked crashes, go see my GDC talk on Forensic Debugging, where I babble about this concept for about an hour. The short of it is, you start with the call stack, and from there you can just start pulling the thread backwards. Most crashes are due to bad data, so anything you can do to validate the data as it's coming in— before it blows you up— will save you an enormous amount of effort down the line.

Concerning general performance, there's one common mistake that I need to bring up. People who use the function `tolower`: Stop it. Just stop it! I've been seeing this used to compare case-insensitive strings a lot, but there is absolutely no need to convert uppercase character to lowercase ones. If you've got a string in your engine, just turn it into a hash to start with. I saw an online multiplayer game spend seven percent of its time just turning uppercase characters into lowercase ones; think about how much money it costs to run that on a dedicated server! It's madness!

Synchronization of Time

Clocks and timers are inherent to video games. For example, look at the update loop and how many frames the game renders every second. Whether to synchronize the multitude of clocks

7. The **Student T-Test** is a statistic test to determine whether a sample set passes hypothesis, the chance the samples are the same or different. For more information see Elan's GDC talk.

in the engine warrants depends on the game. Albert Einstein conclusively demonstrated that it is impossible for observers and different reference frames to agree on a common clock. Valve uses a multiplicity of clocks, and essentially when you're making a networked game you always have to. That's because you've got your client's clock, the other client's clock, and you get the server clock. Occasionally you want to decouple the simulation from the animation, so you'll also have the animation clock, which takes faster or ticks less constantly. All of these clocks are working at different rates.

I don't have a good and complete answer for time synchronization because it depends on the kind of game that you're making, it depends on the way that your animation system works, and it depends on your circumstances. For a game like *Spider-Man*, we actually had to have— for gameplay purposes— multiple clocks, because in a superhero game time slows down and speeds up in dramatic moments. If Spidey smashes through the ceiling, we want to put things in slow-motion, or pause the game when bringing up the weapon wheel. Stuff like that changes how you approach clocks.

Engines Can Change, Little by Little

Adding streaming late in the Insomniac engine is a funny story. Insomniac had made an engine for *Resistance*, and that was used later on for Fuse. When we designed the engine, we went into it with the mentality that our games were primarily in linear indoor environments. As such, we decided the only thing the engine wouldn't support would be open world streaming[8]. So of course, we immediately landed the contract for *Sunset Overdrive*[9]! The point that I would learn from this is that even if you think you haven't designed for something from the ground up, you can get there; you just get there a little bit at a time. The

8. **Open world streaming** refers to the the process of "streaming" (loading) the world/map (sections of the world) into memory while the player moves throughout the world. While the player moves the game decides which section of the map should be loaded into memory, the engine needs to manage memory and framerate during these times of loading.
9. ***Sunset Overdrive*** is a game developed by Insomniac Games for the Xbox One. It is a fast-paced, open world action-adventure, third-person shooter.

whole open world mechanism in *Sunset Overdrive* came from the fact that we had airlocks[10] in *Fuse* to control progress. So essentially, we just said "Well, what if we have airlocks in nine directions but don't actually have airlocks?" Then you just use that same mechanism a piece at a time.

It's often not necessary to go back and rebuild something from the beginning. There's always a step you can take in the right direction. The downside is you end up flooded with technical debt when you do that, but you also end up not having to start over.

Keeping the Team in Sync

When you're working on larger engine teams and want to publicize bugs, there's bug tracking software you can use. Either JIRA or DevTrack are good options; this is a well-solved problem in tech. On the other hand, publicizing features and changes remains a huge issue. The bigger your team gets, the more of a problem that is. Simply saying "we need to communicate better" is a gesture, not an answer.

We do the following things at Insomniac; some of them work and some of them don't. When we make a big usability change and we write a new tool, we will do the presentation on that tool to the people who need to use it. That is somewhat useful in that it's a training scenario, but it tends to go in one ear and out the other. There are also people who you don't realize rely on that tool who won't be part of the discussion. We also write up documentation, but hardly anyone reads it. One helpful thing we do occasionally is record video walkthroughs of how to use the tool. The problem of the video is that it's slow and annoying to watch, but the advantage is that you can follow along with while you're doing it. Think of it like you're cooking and following a recipe.

10. An **airlock** in games refer to an area in which loading of the next chunk/section of the map is being performed. Within this zone the next and previous sections can both be in memory, or with limited memory only the next is being loaded into memory. In the first case the player can enter both the previous and next zones (once loading completes), however in the latter the player can neither advance or backtrack their game's progress while the loading happens.

As for communicating inside the team, Insomniac has a team of about 25 people, so it's a little bit easier. Our former core team director Mike Acton had us do something to get us in the right mindset for every weekly department meeting; someone on the team would do a changelist review. They would look through the change history and comment on interesting changes on a changelist of another member on the team. The reason that we do this is so that people get in the habit of looking at the change lists that are going into the repository.

My advice about having those domain experts came from an era where people were going from PC development to console development more than they are now, and that knowledge hadn't really percolated out. You had studios that had really good PC engines and workflows who were just new to consoles. It is ideal if everybody in the team knows how to get a console devkit[11] set up and working, but it's also not realistic because it's such specific knowledge. In theory, you could write documentation and tutorials and workflows, and that's helpful as far as they go. However, people are always going to have questions that are not answered by the tutorials. As such, it still helps to have somebody who is an expert in the topic. That doesn't need to be their whole job; it just needs to be the person who's really good at doing that thing.

Onboarding engineers is a really hard problem no matter where you are. This is actually where some of the advantage of having a domain expert on how to use the devkit comes from, because then you can ask the person for lessons. At Insomniac, we try to address this by assigning a new person to fix something that's broken in an obvious and easily fixed way. By starting with that sort of problem, you start to pull on the thread of all the pieces that you need to work at the company. The bigger your company is, the more training you can afford.

11. A **game development kit (devkit)** is hardware different from the commercially available version of the hardware, specialized for development. It will have a way of booting with a development version of the game, and modern developer kits have debugging features for the developers.

With Great Power Comes Great Responsibility

Technology is always changing. The amount of power available to consoles and PCs has grown precipitously. PC's seem to have blown up recently, but consoles just keep getting more and more CPU power and memory. People underestimate the importance of RAM because they've been told it affects the amount of art you can get on the screen. Obviously, mobile is a big change— mobile and tablet games are going to be more and more significant over time. AAA isn't so much in that space yet, but a lot of other people are. In terms of revenue, mobile is a huge chunk of the market now, and of course the ubiquitous availability of networking has totally revolutionized the industry. That's because you can always count on the network being filled with people at any given time. This not only applies for multiplayer games, but also for downloadable content, achievements, and all the other good stuff that comes with being connected.

Engines are going to face a few challenges ahead. One is that consoles are getting bigger and have more memory, and teams are getting bigger as well. That's a deal; the tools have to cope with it and the engine has to deal with loading more heterogeneous assets at once. Mobile, obviously, is gonna be a bigger and bigger thing. The tricky thing with mobile is that tablets are actually catching up now in terms of computing power, but they're not catching up in terms of electrical power. Electricity conservation has become a surprising concern, as well as the different UI that you have with tablets. I think people will start to rely on networks more and more, but I think actually the penetration of networking is pretty slow. This is especially true in large countries like the United States, so it will be a while before people are building stuff around the cloud. The age of cloud rendering is not here yet. VR could be a big deal, or it could not— the jury is still out for that.

Interview conducted November 3, 2018.

Going Beyond the Books

Shanee Nishry

***Shanee Nishry** is a Google engineer by day, game developer by night and a fencer in-between. When she isn't working on official Google Daydream VR and AR projects, she is enthusiastically developing her real-time strategy game using her own game engine!*

Programming Origins

My school background is kind of boring. Other than high school, I'm self-taught, so I have no degree or any formal studies in the field. I started a Biotechnology degree, but I didn't finish it because I got into a game studio in the meantime. Maybe when I was about 18 or 19, while I was in my Chemistry and Biotechnology courses, I started studying C++ and I wanted to make a video game. I started teaching myself DirectX, so I used a bunch of books like *Programming Role Playing Games with DirectX* by Jim Adams and *Game Coding Complete* by Mike McShaffry. Both are really awesome books that I'd definitely recommend. I started making my first engine by following these books because I knew absolutely nothing, but these two were brilliant and gave really good example code. By the end of following them, I went from

having no idea what I was doing to kind of having an idea what I was doing!

Because I started programming game engines quite a while ago, I can't think of what was specifically the most confusing aspect. Truth be told, back then I literally knew nothing about game engine programming; I didn't even know how to code properly! I remember one of the first books that I used for C++ was called *Beginning C++ Through Game Programming*. I just followed the books and adapted its code to do what I needed, so the process was pretty simple.

Keeping it Minimal, Decoupled & Data-Oriented

There are a few things that have changed since I started in the games industry. One new paradigm that is very common in the industry right now is data-oriented design. Before that was introduced, everything was classes and hierarchies. Another recent question is "how do you make the systems in your game interact with one another?" So if you have input and you have your graphics and animations, you have to figure out how they will interact in a meaningful way. You also have to determine what gets access to the data. There are so many different paradigms to handle those questions. Some people will tell you to send an event whenever your position is changed, but then when you do that and learn about data-oriented design, you figure out everything is going in your cache and sending an event for every entity. It's just a mess.

When working on *Super Sam Adventures*, we had to figure out how to access the transformation for an entity from different systems. We were originally using a map to access entities and then get the component out of them, and we noticed that on Android and iOS at the time, that map was so expensive because every lookup resulted in cache misses and we were doing too many lookups in the map. It just ruined the frame rate, so we had to change that.

All of these questions about access, interaction between systems, and what a system actually needs to do were probably my

biggest questions after I had a little bit of experience in game development. *When you know nothing, you just follow some book or tutorial, and it all makes sense because you're doing what the book is telling you to do. When you start doing your own thing, though, things don't work as naturally.* So maybe the biggest question I've had since then, and in some ways I (and other people I know) am still trying to answer it today, is about the interaction between systems; what is the responsibility of a system and how much should it be engineered?

I feel like the general consensus these days — and it might change a few years from now — is that a system should be as small as it can be and do the minimum that it should do. The interaction between systems is a very dependent subject, but my preference is that it be as direct and as minimal as possible. By minimal I mean getting all the data that you'd need in one function, if possible. So for example, if I have a transformation system and a rendering system then my render system needs to get the transformation for its entities from the transformation system. Instead of doing it for every entity each time, if you can just make a synchronization point in your engine that says "now I'm going to transfer all the data for the render system", then that's probably better. Having specific synchronization points like that allows you to heavily specialize or multi-thread your systems.

At the same time, I'm recruiting all of my threads to just fill up these few arrays of transformations, which will then be filled in the layout that the render system wants them to be. The render system then just takes the data and does whatever it does, and the rest of the engine is free to do whatever. That way, if you have any kind of multi-threading but you only have the synchronization points, there's no risk of a collision between threads because all of your data has been copied at a very specific point in time. That's the main idea: Make the systems as small as you can and interact as direct (but as little) as you can. One problem that you could find from this is having different implementations of an API and dealing with changes to that API, but it's a different problem which you might be able to avoid entirely.

Developing for an iOS Engine

The *Super Sam* team was very iOS focused. I felt like I was the only Android person in the group. Most of the people did not put considerations into cross-platform development, but I did and I feel like it paid off, because the game was eventually ported to Android and didn't take too much work. At the end of the day, I feel like as long as you keep the very specific interaction with a platform abstracted from the rest of the engine, then it shouldn't take too long to port it to a similar platform. I think that the most difficulty comes when you have a PC game and then you want to port it to mobile or console, because that brings two big issues to the table. One of them is the performance and the memory requirements that are often different, and the other problem is just the input — moving from keyboard to controller or from touch input to keyboard and mouse.

One of the special considerations we had to have when developing for mobile was binary size[1]. And that's partially why we developed our own physics system. We didn't want to link with big external libraries because we wanted to keep the binary and final app size extremely low, and people simply didn't install heavy games. The expectation is that a person would install a game using mobile network which could be slow and limited. Additionally, the Google Store and AppStore would show a warning when trying to install a game over a certain size and most people would simply not install when the warning showed up. One of the reasons not to use Unity back then was because Unity on its own added quite a few megabytes, and that was a big no-no.

Developing for mobile was definitely an interesting experience especially back at the time between the iPhone 3GS and the iPhone 4. For the upgrade to iPhone 4, Apple pretty much quadrupled the number of pixels and doubled the resolution, but the GPU wasn't much better. Suddenly you render the game on an iPhone 4, which is supposed to be better, but the frame rate is just shit compared to an iPhone 3GS. The other fun part

1. **Binary size** refers to the size of the binary files built from source code and other assets.

of working on mobile was memory, and this is something that is often shared by people who worked on older consoles where memory was so limited. In much the same way, mobile was also extremely limited; you had to make sure that you knew exactly how much memory is used by your app and GPU at any given time.

Caching in on Memory

I feel like the biggest concerns with developing solely for iOS were memory management and performance. We really wanted to hit 60 frames per second and have it very smooth with no frame drops whatsoever. We also had problems keeping memory low so the app doesn't get killed randomly by the OS, which was really annoying on mobile platforms. We had to make some kind of a list of least-used memory so if a block of memory was not being used then you could just discard it. We also had to limit and deny allocations above a certain amount of memory. It involves making sure that everything fits into a very specific and tight memory requirement, and it means that we have to immediately discard anything that was not used when you had to have something allocated.

On the other hand, we would not just immediately discard a texture if it doesn't have references because it might have a reference in five frames, and loading it again would be silly. If you have it in memory and then a new memory allocation needs to be made and that texture is occupying the least used memory block, then you can discard it. That's LRU cache[2] basically. Getting the scene and managing it with all of the assets and managing the level so they fit into that very tight requirement was a very large amount of work, especially because *Super Sam* was an infinite scroller; you'd basically fall down in this specific type of world environment, and then at some point it might change to a different one. We had to make sure that these change areas were as seamless as possible and without any kind of frame spikes.

When it comes to dealing with memory management issues,

2. A **Least Recently Used cache scheme** is a strategy for evicting data from a memory cache based on how recently the data has been accessed.

you should determine how much memory you're actually allocating and whether or not that matches your diagnostic tool. Then you can see how much memory is in graphics, how much of that is textures, and how much is meshes, and so on. Not to mention how much memory is going to your entity component system data. It's a really big question of how much memory you actually need and how strict you need to be in memory management. The game I'm working on right now is for PC, not mobile, so instead of being very strict on memory, I'm just focusing on the way that memory and systems are being used. Because of this, I don't need to track it all in a centralized place. On mobile, where memory was way tighter, it was definitely needed.

Engine API Design is a Thing

I've taken the approaches of both overriding the `new` and `delete` functions and constructing my own APIs for memory management. Even so, I can't say which one I prefer. I guess overwriting it made it easy to ensure that everyone is using it, but it was very implicit and sometimes implicit is not very good. I often find if you're managing memory, then you probably want to know exactly what memory allocator you're using anyway. So just overwriting a global function is not necessarily the best idea.

For example, if I want to do a temporary allocation that's nearly immediately released, then I know I need to access the linear memory allocator (one that is reset every frame) and I probably also know the size, so I can instantiate a specific memory allocator. But if I override the global `new` allocator and then I have a general memory management class that tries to manage all kinds of use cases, it's probably not going to do the best job. It might do a good job, but the best is when the user actually knows what they're trying to achieve and they can access a specialized function or API to do exactly that. Also, the platform memory allocators have improved so much in recent years that if you're just replacing the `new` allocator, there might not be a lot of advantage to it, but if you have a specific use case that needs to be optimized then you should use a memory allocator there.

I'll give a quick example of this: When making systems, you often need to allocate memory blocks for your entity components (i.e. this is my transformation, this is my physics component, etc.) You know exactly what the page size is going to be, so you can just use a custom allocator for that system. You also know roughly how many entities you're going to have, and even if you didn't anticipate and you had to increase the size of the memory block, then you can easily just acquire another memory block and use it for the memory allocator later. Then you can, again, just allocate and release pages from that allocator for the specific use case. That's where I find memory allocators to be more useful — when I know how they're going to be used, and what they're going to do exactly. In that way, the user can specify the memory size they want and where it will go.

Following a similar principle outside of memory API, no matter what engine I'm working on, I try to make an API as small and easy to use as possible. To do that, I do some stuff that other developers would probably kill me for. There is different advice out there on the use of singletons versus dependency injections[3], but generally the consensus is that you should have some kind of a context for whatever stuff that you're using. But at the end of the day, if I am a game developer using an engine, all I really want to know is where on the screen my mouse is! So I literally just have a namespace `Input`, and `GetMousePosition` is a global function in that namespace. Having a namespace housing functions is something that can be controversial, because the function doesn't have context as to which window it is listening to. However, I find it very easy to use because I can easily tell someone to go into the codebase and do `Input::GetMousePosition`, `Input::GetMouseButton`, etc., and structuring it like that really helped my productivity. So I try to figure out the APIs in terms of what is going to be useful and how easy it will be use it. What does the developer need to do, and how will the API accommodate that?

The trouble often comes when your system tries to do too

3. **Dependency injection** is a technique where one object supplies the dependencies of another object. A dependency is something that one object needs to run correctly, and injection is the process of passing one object to another.

much. That's why I advocate making small, specialized systems as opposed to big, beastly systems. For example, if you have a rendering system and inside that you have an `AddWater` and `AddBox` and `AddSphere`, then it can probably be split into a few different systems, each specializing in their own specific things.

Good API design can also help port your game to a new platform. At the end of the day, if you expose all of your engine's functionality then you can figure out a way around it later. So if you have access to the keyboard and mouse input and also to touch inputs as part of the engine, then you don't need to specifically map touch input to a mouse button or a mouse input. You can map it that way, but you don't always want to do it. Often if you do it that way, people will say "hey, I need a way to recognize gesture and multi-touch" and so on, and it will just come and bite you back if you've done premature abstraction. That's another lesson, not to do too much abstraction. As long as you have your own engine layer communicating with the OS and then give in the data with the least abstraction as possible and just hide what OS it is and hide what functionality is, then that will be good enough.

Editing with a Level Editor

For developing a level editor, the best advice I can give is to figure out what is actually needed out of it. Determine who is going to use it and the first functionality or two that they are going to need, and then work from there. It doesn't need to be perfect, it just needs to have the bare minimum of functionality supported. For example, in my editor, I have three main functionalities that I need. One of them is the elevation editing for the heightmap[4], another is the texture editing for the terrain, and the last one is the placement of entities into the world. Once you identify those minimum features, you should be able to implement the editor pretty quickly.

The biggest mistake I learned from that experience is over-

4. In computer graphics and games, a **heightmap** is a texture (rasterized image) where pixels have different meaning rather than representing color. One common usage of heightmap is to store surface elevation data.

engineering; do not make features for the sake of making features, *ever*. This is especially the case when you're working on an engine, where you should always have a use case in mind. It's ideal if you are able to make a small game while you're making the engine, even if it's as simple as *Pong* or *Breakout.* If you can make something like that with your engine easily, then you're doing well!

When I made my very first level editor, it was very tightly related to the game engine itself. That was the first mistake, because it was basically living in the same application and codebase, and so it affected everything else a lot. When making the *Super Sam* editor, it was an entirely different application, and it didn't even share any of the code base; all it needed to know was the level format and how to output that. We didn't even use the engine to render the levels.

The biggest question that we had back then was how to make the levels be supported across multiple games and versions. The level format was used for *Bitter Sam*, *Super Sam*, *Super Sam Adventures*, and possibly other games in the future. Figuring out how to differentiate between different games in your level format when using the same editor was one question, and the other was figuring out versioning if you added a new component or creature. We also needed to be able to work around the different versions of the app that may be installed on smartphones. There could potentially be a mismatch between the version of the game on the phone and the level being downloaded from the server, which is completely unrelated to the update mechanism on the phone. If the level has something like a new monster in it but its texture isn't in the application, then things don't work. For those reasons, compatibility support was really important back then.

The level editor of *Super Sam* was using a different framework to render the things in the editor as compared to the game. Personally, I want to match my level editor renderer with my engine renderer as closely as I can. I love it when you have "what you see is what you get", and also when you can play inside the editor. However, that's possibly over-engineering at times and

brings in too many additional features. For my current level editor, I am using my game engine, but I'm making it in an entirely new application and only linking to whatever code that I need to. For example, right now I'm not linking to any of the gameplay stuff, but only to the rendering frameworks and the input and so on.

Using the engine's render in the editor you get the same look and feel, you get the same kind of performance, but it's not the only option. If you look at games like *Warcraft* and *Starcraft*, they have a level editor which is not using their engine's UI; it's using just generic OS UI. But they have a viewport[5] that draws the game window into it, and even in that viewport they added functionality for things like zooming out further in the game. That's just an example of you having all the UI in your game engine, or mix and match; you can use OS functionality to make UI easier to do, while also importing the entity rendering and terrain rendering from your engine to do that.

After-Hours Game Engine

For my own engine architecture, I find that the biggest challenge is rendering large levels efficiently. My solution to that is chunking the world into different segments, which will let you immediately discard thousands of objects because they are off-screen. Another challenge is AI. My RTS is atypical because you can't control every unit; each unit has its own priorities and goals it wants to achieve in life (like making a video game!). I'm using a goal-oriented action planner[6]. So in the beginning, when having only a hundred entities, there's not much of a problem. But then, as you get to thousands of entities, you have to manage to keep some level of detail for the AI. You can't just make the AI stop functioning because they are off-screen; they are still alive and progressing. That's an interesting challenge that's so far removed from rendering, which I feel like is my main exper-

5. In the context of games, a **viewport** is a region of a 2D rectangle that's used to project the 3D scene to a virtual camera and thus provide a way to view the 3D virtual world.
6. **Goal-oriented action planner (GOAP)** is an artificial intelligence system for agents that allows them to plan a sequence of actions to satisfy a particular goal. For a detailed explanation, visit http://alumni.media.mit.edu/~jorkin/goap.html

tise. I can't just say "If you don't see it, it doesn't exist." When a tree falls in the forest and nobody is around, it still exists!

When developing a system I try to keep it as minimal and specialized as I can. For example, I'll do something like a quadtree[7] division with rendering, and that way it's really easy to discard entities not present on the screen. Similarly, using this quadtree division, I can put it into the AI and just update them at a different level of detail. That's the main consideration that I've had with my engine. Other than that, rendering things that scale terrain is a lot of fun. Just using CDLOD[8] for the giant terrain.

If I remember correctly, the way I handled serialization in *Super Sam* was to just give an entity minimal transformation stuff like position and rotation, and then a tag of what entity it is. With my RTS, I take a similar approach where I have a blueprint that all entities can be instantiated from. In my case, I would have a footman or a werewolf blueprint, and then I can just say that an entity is referring to this blueprint while it holds its own transformation in the world.

An interesting thing which I still haven't decided what to do if I need to have a unique entity for a level. The current idea I have is to add an additional blueprint definition that is level-specific, and then when instantiating that special entity it will still use the same system saying "this is my position and this is the blueprint header file and that will have all of the data." I'm still not sure about this approach, because what if I went to add, say, a wounded soldier somewhere? Do I need to make a special blueprint for him? That's a problem that I just haven't reached yet, and there are different ideas on how to solve it. At the moment, I'm using JSON[9] to serialize stuff. I find that it's really useful if

7. **Quadtree** is a special type of tree data structure used in spatial partitioning. It recursively divides the whole space into four quads of the same size, and keeps doing it until each leaf quad contains a certain amount of actual spatial units (like polygons when used for rendering, and colliders when used for collision detection). If you are interested in learning more, refer to the Spatial Partitioning chapter in *Game Programming Patterns.*
8. **CDLOD** is short for the paper titled Continuous Distance-Dependent Level of Detail Rendering Heightmaps. It describes a technique for GPU-based rendering of heightmap terrains.
9. **JSON** (JavaScript Object Notation) is a lightweight data-interchange format

you do not have a binary format when you're just starting with things. If you're just getting to serialization, find a format that is easy to use and is human-readable, and only when you need to publish a product make it into a binary format that is small and specialized.

Being Your Own Product Manager

If you're doing a personal project at the same time you're working a full-time job, you basically have to be your own product manager. You have to set goals both for your time at work and your time at home. You have to say, "okay, I'm going to work and I need to get to these milestones. Today I'm going to shape this feature." And then you focus on that specific feature. But then when you head back home, it's really easy to just lie down in bed, watch Netflix, and neglect your project.

The only way to prevent it is not by gathering motivation but by being persistent. You have to be determined to spend at least five or ten minutes a day on your personal work. Figuring out where you left off can be hard, though, so be sure to keep notes so you can remember what is most important to be working on at the moment. From there, determine the minimum tasks you need to do and then just sit down and do them, and the motivation will come as you do it.

Specialized Engines Aren't Going Away

Many companies are making very good engines and improving their existing engines. I think we can look at Unity as an example here, because they have a very easy to use and commonly-used game engine, but even now they are making huge modifications to it. They have the Scriptable Render Pipeline[10], and they are actively developing a new entity component system. With all that, we can see that even the big engines have to

that can be used for a database. It features a set of syntax that's both easy for human to understand and for machine to parse.

10. In Unity, the **Scriptable Render Pipeline** (SRP) is an alternative to the built-in pipeline. With the SRP, developers can control and tailor rendering via C# scripts. This way, they can either slightly modify or completely build and customize the render pipeline to their needs.

change. I think that's a sign of why specialized engines still exist; it's really difficult to make a generic engine that has everything.

On the one hand, it's awesome because they're really making game development more accessible — I know so many people that are not going to make a game engine but can just jump into Unity or Unreal and publish games. On the other hand, with big engines you're going to run into problems, like what happens when I need to have a million entities on the screen and the engine doesn't give me instancing[11] solutions. We still see a lot of specialized engines, but the big engines are likely to improve and get better in performance. The best part about them is that they have those generalized tools that everyone likes and can use. People can also overwrite and optimize systems in the engine where they need. So I think that what we're going to see a shift in big game engines that allow the user to access more low-level data, but also allow them to do stuff at a high-level if they have no idea what to do at the low-level or want to get things done quickly. Smaller game engines will still exist, there will always be control freaks like me who like to do their own thing.

Interview conducted November 11, 2018.

11. From Wikipedia: Geometry **instancing** is the practice of rendering multiple copies of the same mesh in a scene at once. This technique is primarily used for objects such as trees, grass, or buildings which can be represented as repeated geometry without appearing unduly repetitive.

Thinking About the Data

Martin Middleton

***Martin Middleton** is the CTO at Funomena, an independent game studio in San Francisco he co-founded in 2012 with Robin Hunicke. Funomena has put out award-winning titles on a variety of platforms, including* Luna, Woorld, *and the upcoming* Wattam. *Previously Martin was an engine programmer at thatgamecompany, where he worked on* Flow, Flower, *and* Journey.

Pipeline of Code Optimization

When I started engine programming, the most challenging aspect for me was probably developing core performance. Back in school while I was learning, the best practices for performance were heavily object-oriented with a lot of abstraction. My most useful classes were Electrical Engineering, which is especially beneficial for developing on consoles like the PS3, which has really limited resources and requires you have a deep understanding of what the actual hardware is doing.

An example of this would be thinking about your memory usage so when you assign a variable, where is that value actually coming from? Is it in main memory, is it a local cache, is it

already in the register on the CPU... There's an order of magnitude of speed difference between all those different layers, and it can be really easy to ignore that because most programming languages don't really make that explicit. Overcoming this challenge means thinking more about it for yourself. Whenever you're writing code, you have to internally plan out when you'll be loading up certain values or making sure that the data sticks around in local memory long enough for you to use it. You might also be thinking about what else the processor can be doing while it's waiting that memory to be loaded.

Console platform-based design is very low-level stuff; there aren't any console architectures similar to the PS3 anymore. However, everything is multithreaded these days, so that's useful knowledge I learned from PS3 and SPU[1] programming that has served me no matter what type of engineering I'm doing. Figuring out how to shuffle data between different parts of the hardware so that you're splitting up this computation, "Do I do this on the CPU, do I do it on the GPU, how do I transfer the data between a CPU and the GPU, and what am I doing while the data is transferring..." I think all that stuff is useful, whatever the hardware platform is.

In a certain way, the PS3 was ahead of its time because all of the technology was going in that direction anyway. So it's just sort of the early proving technology, where people learn how to do things that way. Unity's new component systems are structured really similarly to how PS3 engines were structured; we were focused on batching and pipelining things into really small chunks of code that just reads through. It was all about structuring everything to process buffers of data as easily as possible. I don't think it's a coincidence that Unity hired a bunch of senior engineers from Naughty Dog and Insomniac...

At thatgamecompany, I was doing a lot of SPU intrinsic programming, which is a subset of C++. You would use functions that utilize assembly commands, which would tell the processor

1. A **Synergistic Processing Unit** (**SPU**) is the processing unit associated with the Cell architecture of the PS3. The PS3 had seven as part of its hardware, only six of which were usable by game developers.

what exactly to do. However, the problem with this is that you're focusing on one specific problem, and it makes your code brittle. If you need to change that code later on, you have to undo a lot of what you've already done.

But there is sort of a halfway point; if you can get into a mindset where you're always thinking about memory usage, that's something that you will benefit from across the board. Writing code in that style makes it very straightforward, so writing up optimizable code is something that always pays off. Usually that involves being really explicit about when you're loading or writing data. That's not really abstracting things away too much so there's sort of no "magic." If something happens automatically, or "magically," it's usually very suspicious.

It's also more about knowing what exactly is being allocated, and then in what order code it's being updated, not whether it is object or data-oriented. John Carmack has this recommendation that you step through an entire frame per game and step into every single function so you can experience like the entirety of everything that happens in a frame. That takes a really long time, so designing your engine in a way that makes that possible is a good methodology.

Engines Should Guide Games, Not Direct Them

Most engines are built to guide you towards a specific type of game or a specific type of implementation. This is one of the reasons why at thatgamecompany we used PhyreEngine[2] as a framework, we had access to the source code and we were especially deliberate about which features we decided to implement. We didn't want the existing engine to influence how our game progressed. If you're regularly fighting your engine, then you start to question the point in using an engine. You might as well use something a bit lighter, like a framework.

So there's always a trade-off when it comes to how much of

2. **PhyreEngine** is Sony's game engine that is freely available for PlayStation developers. The engine is compatible with the PlayStation platforms of the last decade.

this engine you can use versus what really needs to be side-tracked. With Unity, one example would be the update system; the engine doesn't give you explicit control over the update loop. You have a script execution order, but the `MonoBehaviour` system can be really heavy, especially if you have a lot of objects. Often you end up writing your own entity in a really lightweight entity-system[3] and writing your own explicit update system so you have full control over that.

Custom Entity System in Unity

For our entity system in Unity, we were trying to solve two separate problems. One was when we'd have a whole bunch of objects that are represented by particles or not tied to a mesh. Having 100 game objects is really expensive, so instead we'd just turn that into a really lightweight class and separate that from the game object hierarchy, which makes it very specific to a system. The other problem is controlling the updates. A lot of game objects don't necessarily need to update every frame, and I think Unity has optimized this a lot, but when we first started using Unity, there was just a really big overhead to having even an empty `MonoBehaviour`[4] with nothing implemented. Being able to explicitly have an update loop and dictate which objects are active and which functions they own is good for optimizing, as opposed to having to send messages in every single object to see if it has a handler or not.

For *Wattam*, we're using both our own entity system and Unity's `GameObject` system. The way it works is, by default, we will start with something derived from `MonoBehaviour`, and if it turns out that we need a lot of those "residents" in an array at a time, then we'll decide they don't all need to have their own transform in the scene graph[5] and I can just give them a "simple transform", which uses a `Vector3` for the position and the `Quaternion` for

3. A pure **entity-system** is similar to a flat hierarchy, where the entities hold data and functions to be called by the system manager.
4. **MonoBehaviour** is the base class within Unity that all components which attach to `GameObject` must derive from, it has methods for start, update, and destroy, among a ton of others.
5. The **scene graph** of a game engine holds the entities and components (including transforms and parent hierarchies) of a level, also known as a scene.

rotation. I think that's actually my biggest gripe with Unity, that in order to just store a `Transform` you have to hook it into the scene graph, which just makes everything really slow and heavy. So I think having a lot of objects where you can write out their position and rotation without needing to be plugged into an update loop is really good.

Thoughts on Unity and PhyreEngine

While Unity is free for non-professionals, I haven't found it to be as accessible as PhyreEngine. Phyre was available for anyone who was a PlayStation developer; you could just download it from their dev forum. Sony also made Phyre's source code available so you can modify it, whereas Unity is much more of a black box.

For professional development, Unity is actually our second-highest cost in software, coming only after Maya. Unity has a somewhat pricey monthly subscription fee as well. On the other hand, Unity is much more widely used while the Phyre team was very small within Sony. They were very limited resources-wise, whereas Unity is an enormous organization with lots of engineers. That being said, Unity had not prioritized console development at all for a long time; they just didn't see that as their audience. Developing a console title with Unity was kind of a struggle, and still is in some ways. They're just now starting to support console development more, I think because the platform holders themselves are investing resources because they know that a lot of people use Unity.

Technology from *Flower* to *Journey*

A lot of the *Journey* engine was the *Flower* engine. We started developing *Journey* with the *Flower* engine pretty early on because we really wanted to focus on an engine we could iterate on and develop and use for multiple projects. The structure of it was meant to support multithreading really well, and having systems for gathering up data and sending them off to the SPU's and synchronizing and reporting back when those SPU's were finished with the data.

Animation support needed to be built for *Journey*, because the *Flower* engine initially didn't have much, since most of the animation in *Flower* was procedural. *Journey*'s sand system also evolved from the grass in *Flower*; I kind of started off by using the structure of the grass system. In that game, the grass was our test for SPU usage, so any time we had extra SPU resources we would just give it to the grass system so it could render a little bit more grass, or push out the LOD[6] a bit more.

The other main system we had to add for *Journey* was networking. Going from a non-networked game to a networked game is a pretty big shift, because all of a sudden, all your important objects and game events have to be serialized[7] and addressable with IDs. At that point, you can't just store a list of pointers anymore, because those pointers aren't going to work across machines. You have to think of a higher level way of referring to them.

Switching those references from pointers to an index/ID-based reference system was something that was more challenging than expected. Another big challenge was figuring out how many players to support beforehand. *Journey* was peer-to-peer[8], so one of the clients acted as the server. Initially, we wanted to support four people at a time, but that wound up being too complicated and out of scope for us to implement. That changes the design a lot, since scenarios meant for four people needed to work for two people. Figuring out the specifics of multiplayer is good to do early on in the development process.

There's also the whole aspect of synchronization between machines. We had two different update paths: One was for content that relies on the other machine to know about, and then the other for things, like particle effects, that can be done locally.

6. Level of Detail (**LOD**) is the process of simplifying a model/mesh by removing vertices and detail. This typically occurs when the model is far enough such that the details are relatively insignificant compared to others in the viewport.
7. **Serialization** is the process of data being converted into a byte stream for easier storage and transfer, think of it as similar to a save and load system.
8. **Peer-to-peer** networking is where every machine to one another, which requires more bandwidth per client and more complex data authority handling but avoids needing a dedicated server. Peer-to-peer is generally harder to implement than client-server.

If you start off with that in mind, then it gets a lot easier later on. Otherwise, it can be hard to keep track of what's being synchronized and what isn't. If you start to try to synchronize something after the fact, then you have all these side effects that you weren't expecting, which often leads to many other things you have to synchronize as well. It depends on the authority[9] and on how accurate you need to be. For example, in a competitive multiplayer game where you are targeting something, which player's machine decides if the shot actually connects? Then waiting for the other machine to agree with you can sometimes take too long, so you have to start reacting to what *you* think happened, and then be able to back out of that if it turns out that both machines don't agree.

Journey's Peer Networking System

For *Journey*'s networking, the peer-matching system was based on the lobby system that Sony provided. The way it worked was levels were split up into grids, so depending on which grid cell you were in, you'd join a room that was sort of like a hash[10] with the grid cell and their specific game state. Once you join that room, everyone in the room gets a message that you joined and they send you their data. It compares the two players' pings and what game flags have changed, and from there it determines whether the two people are compatible.

One interesting aspect to this is that we implemented a maximum room size — otherwise the room could be flooded by tons of people spamming each other with messages. If a room becomes too big, then you create a new room. That means the number of rooms grows as the audience grows. But the game's online play goes through periods of more and less activity, so then you have the issue where there are a whole lot of rooms but there are only a couple people in each room, and they're all stranded from each other. To solve this, I ran a room defrag-

9. **Authority** with regards to networking is when a certain machine, typically the server, has the control (final decision) on the state of a variable/script/entity.
10. A **hash** is a structure that maps keys to values through a formula defined to convert structures into an index, typically the formula is constructed to avoid collisions between similar objects. The hash of the same object will always return the same value.

mentation[11] system where if a player is in one room for too long, they would leave and try to join a busier room.

Matchmaking was one of my biggest fears in the development process. We had been talking to other developers of online games at the time, and they said when your game first launches, there's a big spike in players but that often tails off over time as players move on from your game. You have to design for two online environments: The popular and unpopular. On *Journey*, we planned for both early on, but the fact that people were still playing six months after launch was nice to see. We weren't expecting that because we had heard how quickly online player bases drop off. Even years after, I was still able to show somebody *Journey* and managed to connect with people in-game, which really makes me happy!

Versioning & Deploying Tools

At thatgamecompany, we initially developed web tools for two reasons. One was the restart game issue — we wanted our developers to be able to live update values, but developing the editor into the actual engine itself would have been way too hard. So we knew we needed some kind of remote editor. Our second reason was the fact that UI for web is immensely easier, especially at the time. I think web tools can be a pretty good system, but the main issue is that once you get to a certain level of complexity, it becomes much harder. I don't see a really advanced complex animation tool like Maya, for instance, being deployed on a web platform anytime soon.

With executable/desktop tools there were some deployment issues where I'd add a new feature, and it would take everyone on the team a while to be using that change. For that change to propagate, it would get deployed on people's machines on the PS3 side. When you're developing any type of networking, early on you learn that you need to implement a version-num-

11. **Defragmentation** is the process of reducing fragmentation (well that's a dumb definition), where **fragmentation** is where memory is used inefficiently with lots of gaps in between used memory chunks. In this context, defragmentation is used in the sense of keeping the rooms fully utilized, not wasting space on a fairly empty room.

ber protocol[12] that doesn't really change so that it's backward-compatible. If you have this version, in a worst-case scenario you can just ignore any messages that are coming from an older version. On the website with our tools on it, I like supporting several older versions at a time so that my teammates can download the new PS3 code and update it on their time.

Implications of Prototyping

At Funomena, we are still figuring out how to handle prototyping phases. The platonic "ideal" is that you handle prototyping separately, and then everything is locked down and you implement it. Unfortunately, it never really works that way, because after you've invested a lot of work into a system, it's tough to start from scratch all over again — especially since it's around that time when people are really itching to see a more polished demo. For certain prototypes, I definitely encourage doing that in entirely separate projects so you're working in a different codebase. Maybe you branch it so that you can clean it up and try merging it back in. If you take the path of least resistance, that code will eventually make its way back in, so don't be too sloppy with it.

It can be challenging to make the separation between your different feature implementations. With *Journey,* for instance, a lot of systems were tightly interconnected with each other. That made it hard to design a specific thing in isolation because it was depending on all these other things as well. In that situation, I think what you do is you get more comfortable at dealing with the technical bit and foreseeing what problems could arise when combining two things. The trick is to be really explicit about your assumptions in the code, so down the line you'll know to change it if those assumptions are no longer true.

The Spectrum of Engine Development

I think if you're learning about engines, one way to do it is to

12. Each layer of the network stack has a duty, but they can follow those duties in different ways, and the different implementations of a layer is labeled a **protocol**.

start with what specifically you want to learn (i.e. how application context[13] works in Windows), and with other things (i.e. graphics rendering), you can find a lightweight framework to use. Then, over time, you can replace it bit by bit with something you make yourself. In this way, you can make something fully your own. I have also done a lot of audio programming on games that I've worked on, but I never wrote an actual audio renderer[14]. I always relied on either SCREAM, which was Sony's audio library, or Wwise, the industry standard. At some point, I think it would be fun to write my own audio code, but it's already written and more importantly most of the sound designers that we work with are comfortable with Wwise, and so it doesn't make sense to mess with that.

Unity is far on one end of the spectrum, and then at the other end is the Casey Muratori "write everything yourself" method in *Handmade Hero*. There's a lot of in-between as well; you don't have to go so heavy-handed as an engine. There's SDL[15], or graphics wrappers where you just write forward to the wrapper and it handles the underlying OpenGL[16] or DirectX[17]. There are also lighter frameworks that bring in bits and pieces like meshoptimizer to build a library — things you wouldn't have the time to make otherwise. I think that's a really good way to learn.

Interview conducted October 3, 2018.

13. **Application context** is the context, the set of data required to interrupt and continue a task, of an application.
14. An **audio renderer** is a system which plays/outputs spatialized sound, sound that is positioned in the world.
15. Simple DirectMedia Layer (**SDL**) is a hardware abstraction layer for audio, input, and graphics across multiple platforms.
16. **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.
17. Microsoft **DirectX** is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming, on Microsoft platforms, like Windows and Xbox. It is most known for Direct3D which is the graphics API used for creating windows and rendering, and serves similar purposes as OpenGL.

The Engine Sandwich: Made with Super Meat

Tommy Refenes

***Tommy Refenes** is the programmer for the million-selling and award-winning* Super Meat Boy *and the lead programmer on the upcoming* Super Meat Boy Forever. *He also appeared in* Indie Game: The Movie.

Compartmentalizing is Key

Regarding my first attempt at a game engine, I experimented with making a game engine right before I started working in the games industry. I got really confused while working on it, but it was really helpful going through the process. It prepared me more for working on the 2x engine[1] and then porting that engine to the Xbox 360 and working on the 360. By the time I left that first company, I learned the benefits of engine abstraction[2] and trying to keep platform-dependent[3] code sep-

1. **2x Engine** Refers to Unreal Engine 2.x, which was originally debuted in 2002 with America's Army. The Unreal Engine is a source-available game engine developed by Epic Games.
2. Engine abstraction is the part of the engine code which depends on the hardware/software platform that the engine runs on and will be different on each platform. For example, the code that talks to the operating system on macOS

arate from game code. This is important because in the first engines I would write, everything was all just one project. It was just a quick and dirty way to make something that worked. These are the places where you have to compartmentalize the different parts of development. For instance, the way my current engine is structured, there is no concept of Xbox, Switch, or PlayStation in the code of *Super Meat Boy* or *Super Meat Boy Forever*. There's very little concept of platform even in the engine code, which is the piece that actually talks to the platforms.

The biggest point of confusion when I was starting out was actually learning how to properly abstract and compartmentalize and implant, because when you're at the beginning of making an engine, *where the hell do you start?* That's a huge challenge; even getting something simple like a triangle drawing on the screen takes 100 lines of OpenGL[4] code or 200 lines of DirectX[5] code. From that point, even when you have a triangle on-screen, where do you go from there? I think learning that I needed to compartmentalize, having: my asset loader, my game, my controller code, my audio code, having all those things… I think that was the biggest point of confusion for me. Being able to compartmentalize those systems actually allowed me to have that starting point.

Basic Principles of Abstraction

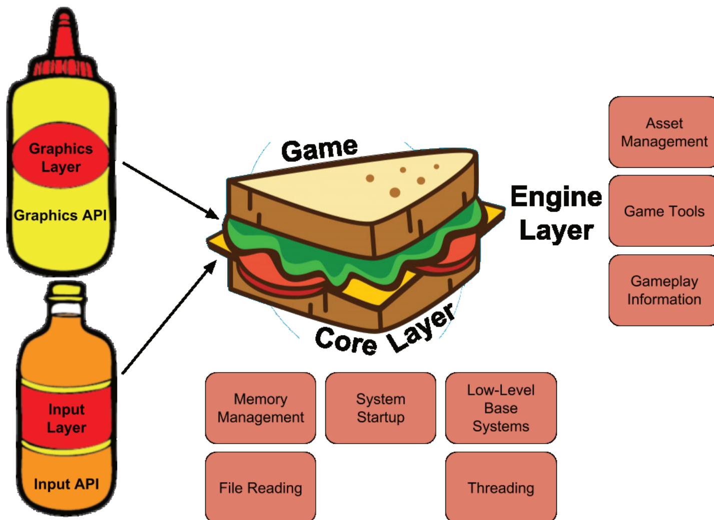
Right before PAX last year, *Forever* was only running on Switch but two or three days before the convention, I realized I wanted

will be different from that on Windows. Engine developers usually tackle this problem by having an abstraction layer on top of operating system code. So the code above that layer still looks the same when you swap out the underlying operating system.

3. **Platform-dependent code** is application code that is dependent on one operating system, and typically won't run on multiple.
4. **OpenGL** is 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.
5. Microsoft **DirectX** is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming, on Microsoft platforms, like Windows and Xbox. It is most known for Direct3D which is the graphics API used for creating windows and rendering, and serves similar purposes as OpenGL.

to be able to show it in 1080p because I only had the handheld Switches to show at our booth. At first, I thought I should just buy some computers, but then I looked over on my desk and saw my two Xbox dev kits. So I ported to Xbox in about two days, and then PlayStation in a day and a half. I attribute this quick turnaround time to how the engine is structured. If it weren't for abstraction, porting games wouldn't have been nearly as easy for me.

The engine has a core layer which consists of memory management, threading, anything specific to system startup for consoles, file reading, and all the very low-level base systems that will change per platform. The core layer is on the very bottom. On top of the core layer is the engine layer, which sits between the core and the game (which is at the very very top). The engine layer handles stuff like asset management, including the SWF[6] reader and Spine[7]. It also contains gameplay information, such as levels and the base level of your characters and players.



6. **Small Web Format (SWF)** is an Adobe Flash file format used for multimedia, vector graphics and ActionScript. SWF files can contain animations or applets of varying degrees of interactivity and function.
7. **Spine** is a 2D skeletal animation software for video games by Esoteric Software.

You build upwards from the engine layer to make your game, but it has all the tools for making a level, like placing tiles from assets. If you imagine the engine layer is sitting there as a sandwich between core layer on the bottom and game layer on top, out to the side you have graphics, audio, input, etc. All of those are abstracted out, so if I want to load an asset I go into Tommunism Graphics (the namespace and the name of my engine), and then access Texture and from there Create Texture. All of those exist in-engine, but that function itself— the low-level Create Texture function— exists in the graphics layer that talks directly to the API for that particular platform. On the PC, for instance, the graphics layer is DirectX 11, and DirectX 11 has all of these functions that are defined like: Present, Draw, Load Texture, Load Vertex Buffer, and Load Shader. The engine layer can call those, so when I compile the static library of DirectX 11, it links to the engine. The engine layer only cares that Draw, Present— all these things that I need— are defined in this lower level of the graphics API. It was easy because I wasn't porting the engine or the entire game; I was simply porting graphics and input.

If you abstract to the point where the engine can load in textures and vertex buffers and shaders, that's pretty much all you need. So porting those libraries, those static libraries of inputs and graphics, audio and whatever else, plus core which is the lower level stuff with file reading and everything, is all that is needed. None of those libraries talk to each other; they all talk through the engine layer, and the gameplay layer talks to the engine layer, which means that when I ported the Xbox version, I only had to port about three libraries.

Once I ported those three libraries, as soon as I boot the game up, it works. Sometimes a couple little weird graphical things will come up because DirectX 11 works differently on Xbox than it does on PC or PlayStation 4, so I have to switch some shader stuff. But every time I port something, I get something on the screen immediately because everything is abstracted and the engine doesn't care as long as functions are defined.

Using 3rd Party Libraries

I like understanding what's happening in my engine at all layers. All of the of the render pipelines stuff is custom, the layers that talk to the Xbox One DirectX 11 API/PlayStation/Switch and any other platform's graphics API are all written by me and then put under my abstraction layer for graphics.

One of the big 3rd party libraries I used was Box2D[8]. When I looked at the Box2D code, I thought it was good and I understood how it worked, but didn't want to write it because I didn't want to go through the process of writing a physics engine[9] that can do exactly what Box2D does. That would take iterations of velocity and position solving, which I'm capable of doing but wouldn't want to waste time on. What I did was modified Box2D to work with the physics in my engine by doing the physics solving, acting as a solver.

I think the biggest part of game development is the asset pipeline. With Flash, a lot of the asset pipeline is just taken care of. In addition to Box2D, there are some other third party libraries I use. I integrated FMOD[10] for audio, because I originally made my own audio engine for the first *Meat Boy* but didn't want to do that again moving forward. I also used Bink[11] by Rad Game Tools, because cutscenes were getting a little too crazy. *Forever's* black-and-white intro cutscene is rendered as vector, but everything else is actually movies now because it was just easier for me to edit the movies and add effects than it was to put them in Flash. That made life way easier. The way Bink just plays videos makes things so easy, because all I have to do is integrate a few things and then movies run on every platform. Overall, it's a much better use of my time and money.

8. **Box2D** is an open source C++ engine for simulating rigid bodies in 2D. Box2D is developed by Erin Catto and has the zlib license.
9. **Physics engine** for games usually consists of two parts: collision detection and collision resolution, and solver refers to the resolution part. Collision detection detects what objects collide with each other first, and then the solver determines their correct physical response, like position, rotation, velocity, etc.
10. **FMOD** is a cross-platform audio engine and authoring tool used throughout the game industry. It was used by over 2,000 games in the last 15 years.
11. **Bink** is the defacto video codec for games created by Rad Tools.

Customizing toward Flash

The way my engine is structured, there is only one concept of Flash and that is the SWF file loader. Everything else goes to a higher concept of animation and instance, so while I have support for Flash, I also built-in support for Spine and .obj files. The engine doesn't care; for instance, some of the things in *Super Meat Boy* and *Forever* I know are going to be Flash animations because that's where all of our assets are. For *Meat Boy*'s character animations, I have a `CharacterAnimation_Flash` class that is part of character animation, but the Flash would have special little things. This lets me use the best parts of different file types and the engine is able to render them regardless. That way it's not pigeon-holed or anything, it's just a thing that the engine supports. This versatility gives me a lot of options for future projects. Down the road, if I want to make a 3D *Meat Boy*, I can just make a 3D *Meat Boy* with the engine. Or if we were going to do hand-drawn pixel art, which Flash is terrible for, I can do that.

Tools for Your Teammates

My philosophy is whatever program my artists or designers are comfortable working in, I'm going to try to make it so that they can continue their work with that program. I know that on my own, I am not going to make something that is better than what they're used to.

If you think of this with regards to programming, I'm very used to C++, and people always ask me if I'm going to use Jonathan Blow's JAI[12]. And I tell them no, because I'm comfortable with C++. I'm not looking to change it, I don't think C++ really needs to be better because I've worked around the limitations of it for so long that at this point they're not limitations anymore.

When making *Super Meat Boy*, Edmund[13] was the most comfortable with Flash, just like I was, since we both started out on New-

12. **JAI** is a language being developed by Jonathan Blow and his at Thekla to address some of the issues game developers have with the current industry standard, C++.
13. **Edmund McMillen** worked on *Super Meat Boy* together with Tommy as an

grounds[14] and everything. It's what he drew in and animated in. So instead of making something new or trying to shoehorn him into some other process, I took the burden on myself. I made a tool that would let us export all the images, timelines and animations from Flash. The tool arranged all the assets on the stage, and then moved and sorted them so they were packed into the smallest texture possible. Finally, it would export texture coordinates and go through the timeline, which would be used by my custom animation index and PNG files that run all of *Super Meat Boy*. And that made it easy for Edmund to just work the way he needed to work.

After *Super Meat Boy* was done, I thought it was time to cut out the middleman of PNG files. I wanted to make my engine just render SWF files, which I figured has to be possible because Flash renders them. What I did was download the SWF file specs and read through the full documentation. I found out exactly what's in an SWF file and I used that to make the new engine. The new engine can just load in an SWF. I also tied in some native code that I can put into actual FLA[15] files, and the artists can export those to SWF.

This goes back to making it easy for the people you are working with. For example, Paul was making all of the animations for the enemies and they're all vector graphics, so we can scale those to any size and they work in-engine. He doesn't have to jump through any hoops. The only thing he and the other animators can't use is the Flash's filter stuff, but I've supported a lot of the filters that are in Flash now because it just makes sense. The engine renders them fast and it's a nice little asset package that you can zip; it does a lot to help out the team.

artist and designer. He is also famous for making *The Binding of Isaac* and its remake.

14. **Newgrounds** is an American online entertainment and social media website and company. The site hosts user-generated content such as games, movies, audio, and artwork in four respective site "portals".
15. **FLA** is the file format for projects created by Adobe Animate, and can contain graphics, video, text, audio, and more. They are often saved as SWF files to be used on the web.

Pixel to Vector Art

A lot of *Forever* is vector graphics, but a lot of it is still rasterized[16] images that are read and compressed into the SWFs. The only weird challenge that came with switching over to vector from raster graphics is the fact that graphics cards are made to render polygons, and they're actually pretty garbage at rendering textures. That's why you have games that look like the new *Spider-Man* that runs at a steady framerate, but a game that has a bunch of particle effects that are 2D and has high-resolution art can run like garbage. It has to do with the fill rate, and the way you get around the fill rate of the hardware is by carefully choosing which pixels to render. You really can't do that with full-screen PNGs because you have to encode alpha and optimize to make it actually look right. That kind of optimization is not as trivial as doing a backface cull[17], where if I was rendering a character model, I would only render half of it because you don't see the other side.

With the vector stuff, I actually needed to render a little differently. This meant I got to use more advanced rendering techniques like stencil buffers[18] and depth culling[19], which actually makes the vector graphics render way faster than any of the PNGs. It wasn't so much a challenge, as it was a different way of thinking. Everything in *Super Meat Boy* was just layered PNGs; you draw your background, you draw your foreground, you go from bottom up. With vectors, though, you tend to go from top down and mixing the two can be ridiculously inconvenient.

Stealth Loading in Super Meat Boy

Super Meat Boy and *Forever* levels consist of the palette and level information, which then need to be loaded by the engine. Information in *Super Meat Boy* was essentially run-length encoded[20]— all the tiles, object positions, and object properties.

16. A **rasterized image** is one which is represented as a grid of pixels with RGBA color.
17. **Backface culling** is the technique of performing visibility checks on a mesh to not render the back face (face not facing the camera).
18. **Stencil buffer** is an additional depth buffer to the depth and color buffers.
19. **Depth culling** is the process of deciding which elements to render based on the distance from the camera and if it is being hidden by another element.

Those are all very small files that can be read immediately in the memory and then have direct access through memory. When you have direct memory access instead of actual file loading, you don't have to worry about cache misses or swapping memory around to be able to load everything. That's just the level data; the palette data is your textures and your animations—everything that has to do with the visuals of the game. For example, The Forest[21] is a palette, so every level in the Forest uses the forest palette.

In *Super Meat Boy,* when you would go into the chapter, it would start playing a little cutscene where we'd introduce the chapter, and at the same time it would start loading in the palette on a different thread. During all of this the game would pause, but you wouldn't notice it because at the end of the cutscenes it has the black and white screen that says "The Forest" that goes "dun dun dunnnn!" Even if you skip the cutscene during that time, it's continuing to load the palette. Because the palettes aren't very big anyway, it's only about four seconds so that jingle gives the game enough time to load everything I have. That's then cached so when you're in the level the only thing it is loading new is new level data which is anywhere between 5 KB and 50 KB. Even the craziest levels are only about 100 KB. The textures were already loaded from the palette, then the game would continue from there. I utilized a lot with the presentation of the game to actually background load assets such as palettes. I think the boss battle cutscenes loaded either during the chapter intro or when you were on the overworld map getting ready to hit the button to go into the boss level. Again, the loads were small because the bosses were just a couple assets, and the levels and palette were already loaded. So for the Lil' Slugger[22] fight, all that is loaded is Lil' Slugger's animations, which are just him walking. The reason everything seems to load so fast is that all the heavy lifting is done when the player doesn't notice it.

20. **Run-length encoding** is a form of lossless data compression where data is stored as a single data value and count, for more.
21. **The Forest** is the first chapter of Super Meat Boy.
22. **Lil' Slugger** is the first boss in the Super Meat Boy game.

"Garbage Physics"

The garbage physics came about all from iteration[23]. For *Forever*, I wanted to keep the exact same feel of Meat Boy with the new game, so a lot of the physics are exactly the same. The player doesn't have control over the direction, so physics from *Super Meat Boy* like air friction for when you're turning around in the air, the amount of time when you're going in one direction and then switch to another direction, those don't exist in *Forever* because you don't have any control over direction. But how his jump works and the new mechanics, like his punch and dive, were also developed from what I call "garbage physics." There are different obstacles, like fans, that had to be adjusted for this game, which also required iteration. Nothing in the game is physically accurate to real physics, not a single thing. All of Meat Boy's weird controls are done in-game, not in-engine.

Interview conducted September 13. 2018.

23. Tommy talked more about the creation of the Meat Boy physics in an interview with Casey Muratori at HandmadeCon 2015.

A Lost Art

Raymond Graham

***Raymond Graham** has extensive experience working at the bleeding edge of technology. He has over 19 years experience developing 3D interactive entertainment products for various platforms (Xbox One, PS4, iOS, Xbox 360, PS3 and many others). Ray has worked in technical management, leadership and individual contributor positions at several leading Gaming and Entertainment companies, including Ubisoft, 2K Marin, Electronic Arts and Visual Concepts.*

Who is Raymond Graham?

I'm a graphics programmer currently working at Unity, but I've been all over the place. I was born in Jamaica, grew up in Toronto, Ontario, and went to school at the University of Waterloo. Out of school, I worked at NuFX in Chicago on some NBA games, then proceeded to work at Visual Concepts, EA, 2K Marin, and Ubisoft, working on games like NBA 2K, *The Godfather*, *BioShock 2*, and *Splinter Cell: Blacklist* as a graphics and tech lead. I spent some time at Apple working on mobile GPUs, then ended up going to Unity so that I could still help game developers even if I'm not working on games.

I've been involved with Gameheads Oakland, a nonprofit group that teaches kids from high school to early college, who have lit-

tle to no game development background on how to make video games. I'm also part of /dev/color, an organization of black software engineers across all disciplines, but it's kind of weird because I'm one of only two video game engineers in that group in San Francisco; everybody else works at tech companies! It's kind of like a mentorship group, where everyone is trying to help each other achieve their professional goals. It's a great way of meeting more people in software engineering that are like me, and right now there are about 300+ members across San Francisco, New York, and most recently Seattle and Atlanta.

The Console Evolution and Engine Implications

For me, the most confusing part of programming on game engines was understanding how everything fits together. I wondered how a game engine even worked in the first place. The first game I did engine development for was *NBA Street,* where I was responsible for all the graphics work as well as loading assets on disk. I did a really terrible job of it at the time. The game shipped just fine, but I think I could have done a much better job if I had learned about things like disk I/O[1] and how long it takes something to read off disk and into memory. If I were to go back today and do it again, I think I could do it way better.

There weren't any graphics or engine tricks we employed with early basketball games; we just worked a lot. It turned out to be harder than you'd think, because one of the main objectives was working within the memory and performance budgets. Every year the games need to introduce new features, which makes it a challenge to find enough memory to keep pace and fit everything in. What's more, you only have in a year to do all that. That's one of the reasons why I stopped working on basketball games; I wanted more time to try more cool things and research more things and make even better features. There were some really impressive things we did on the basketball games, but they were only important for sports games. Moving on in my

1. **Disk I/O** includes read or write operations involving a physical disk. In general, to load an asset from the disk, the system will need to read it from the hard disk, write it into the memory (and possibly cache), which takes a lot of time.

career allowed me to explore what else was out there and have more learning experiences.

Developing for consoles in the late 90's compared to consoles today was very similar, but also different in a lot of ways. That's because the Nintendo 64 and PlayStation were completely different pieces of hardware with different specs and texture requirements, which meant the whole pipeline of how you actually built your data was different. The Nintendo 64 had a completely different graphics pipeline, but we still tried to abstract as much as we could. For instance, the engine was designed with the gameplay stuff in one layer and then the core stuff in another layer that talks to the hardware so that core layer was kind of the same on all platforms. Nowadays, with Xbox One, PS4, and PC all essentially having the same architecture, I would say the process is a little bit easier.

I think the console makers want to make things easier on the developers that are making games, and so they want their platform to be as easy to program for as possible. With the way hardware is now, though, I don't think we ever will go back to the PS3s' style. The thing about SPU is that it works really differently from every other platform around, and so being able to to do it in a way that's cross-platform and that gives you enough time to actually really get the most out of that platform is definitely a challenge. It's a pity. It makes me real sad, because that's likely the way it's going to stay. It's disappointing because when you look at PS3 exclusive games, like the ones Naughty Dog made; they milked the most they could out of the console. It's an incredible platform to get the most power out of, but it's just too specialized.

I loved working with the PS3 cell architecture! A little bit of background: When I worked on *NBA Street*, PS2 had the VU architecture with VU0 and VU1 chips[2], and with that if you wanted to get the most out of your graphics platform, you would have VU1 basically doing the draw calls[3], batching[4] up the

2. **Vector unit architecture (VU)** is the architecture for the Emotion Engine that was used in the Playstation 2 console. The two processing units were focused for 3D math and predecessor for the vertex shader pipelines.

polygons to send them over to the graphics chip. At the time, that was all a form of assembly language where you would just have to figure it out. I actually had the four black manuals on my desk that I would pull out to determine which bit goes where, and I found that really fun! PS3 and SPU is very similar to that, except its programming language is either C or C++. My brain naturally understood how it's supposed to work: You DMA[5] in and you double buffer[6], and you work on the data as more data is being DMA'd in. From there you stream it out, switch to another buffer, call for the next DMA. Because of this, you can work on batches of data in a streaming parallelized fashion.

Porting between Non-Compatible Architectures

Porting *BioShock* to PS3 was hard, because one of the main requirements we were given was to keep all the data and level loading flow the same. The problem with that is that while Xbox 360 and PC have unified memory architecture[7], the PS3 does not. PS3 has video memory and it has main memory, and you can't use the video memory for general purpose stuff because it was too slow to access it directly. With *BioShock*, we had a game made for unified memory architecture, and we were trying to get it to run on PS3; most of the system memory was graphics related but it couldn't all fit in the 256 megabytes of the PS3's video RAM. So there was this constant struggle of figuring out how to get the memory to fit. The last thing that we could have done (but would have been too much work) was cutting the levels up and adding loading screens. Because there was no streaming at that point, it was still "load level" and that was it. We didn't

3. A **draw call** is a command from CPU to GPU that contains all the information encapsulated by CPU about textures, states, shaders, rendering objects, buffers, etc.
4. Encapsulating a draw call is expensive, and the GPU can render fairly fast, so **batching** draw calls up is a good technique to speed up.
5. **Direct memory access (DMA)** is a technique of computer systems that allows certain hardware subsystems to access main system memory without taking up the CPU cycles.
6. **Double buffer** is the use of two buffers to hold data. By switching the buffers, the reader can see the complete version of data instead of a partially written one.
7. **Unified memory architecture** use a portion of a computer's RAM rather than dedicated graphics memory. It is a single memory address space accessible from any processor in a system.

want to do that because it would change how the player experiences the game, and it would take all sorts of technical work to do that. In the end, we handled the issue by enabling the virtual memory[8]; PS3 had the ability to use its hard drive as a backing for virtual memory, so we used that to fit the stuff that spilled over into virtual memory. It wasn't the greatest solution, and there's definitely some noticeable lag in the final product, but it was the only way that game would have shipped.

After that, we were able to take what we learned when working on *BioShock 2* and were able to budget for the PS3's memory restrictions and do it right. Even so, *BioShock 2* shipped with the same PS3 virtual memory system. We had good intentions, but sometimes you just have to do whatever it takes to get the game done. That was a really hard problem; even at Ubisoft we faced the same issue. On PS3 you have these two different memory pools and then on Xbox 360 you have one, so managing memory in such a different way was a real challenge.

On the graphics side of *BioShock 2*, our improvements were more about making the engine ready to do better visuals on PS3 and Xbox 360 at the same time. We also added a few graphics features here and there to improve the game's look. For example, we added motion blur and implemented Unreal's material editor[9] so that the artists could actually have a proper material editor to make shaders. Previously the artists would have to bother programmers to implement every little one-off shader. All in all, though, we didn't do too many new engine things for *BioShock 2*, since the art style was the same as the original. It was more about finding little places to improve, and also making sure the PS3 version was rock-solid this time. Once that was done we felt confident shipping it.

The Winding Road of an Engine Developer

I decided to go work at Apple because I had spent about 15

8. **Virtual memory** is a memory management technique that abstracts uniformed memory space from different kind of storage device.
9. The **Unreal Material Editor** is a node-based graph interface that enables you to create shaders. For more see the Unreal Documentation.

years in the game industry. After years at video game studios, I didn't really like how the games industry was ballooning. When I started on *NBA Street,* I was on a team of only six programmers. The *NBA 2K* team was maybe 20 programmers, and then by the time I was on *Splinter Cell: Blacklist,* I would be in meetings with over 100 other programmers. As video games got more complex, teams got bigger. Today, it's not uncommon to see a 800-1,000 person team. I was tired of working on those big teams, and I just wanted to do something smaller.

Much like Apple, Unity definitely has a very rigorous testing process, because we make software for millions of people and that makes testing your code essential. I think I brought back some knowledge I had of the actual workings of how the chips on those devices work, which helps us to figure out what the fastest path is. That's huge when figuring out the best path to deliver graphics to phones. Being at Apple helped a lot in understanding how mobile devices in general work, so that broadened my skill set for sure.

PC development is also definitely changing with the times. I think we're starting to see more people embracing low-level graphics APIs like Metal[10] and Vulkan[11]. Pretty much all the console devs are telling them, "Welcome to the party, we've been doing this for years!" On PS2 and on PS3, we were working with very specialized low-level API's that gave you access to the hardware. Finally now that people in the desktop space want Vulkan and Metal, we can tell them why that's important. Also, we're trying to use ECS[12] systems and data-oriented design at Unity, which I think is also something that we *had* to do on the console side to get performance. You had to have your data laid out efficiently to save memory and get performance. Now you're starting to see that be more of a focus in general code, which I think is a good thing.

10. **Metal** is a low-level, low-overhead hardware-accelerated 3D graphic and compute shader application programming interface (API) developed by Apple Inc.
11. **Vulkan** is a low-overhead, cross-platform 3D graphics and compute API targeting high-performance realtime 3D graphics applications such as video games and interactive media across all platforms.
12. **Entity-Component-System (ECS)** is an architectural pattern that follows composition over inheritance principle and is mostly used in games.

A lot of the other features we make at Unity are driven by the artists; if there's something they need to be able to do but don't have a solution, we create a solution out of necessity. As a graphics programmer, your number-one client is the art team. We're just making sure that they have all the tools they need to actually use the system.

Working with artists and developers as part of the Spotlight team is kind of a mix of things. Most of our work is either implementing features for teams and then those features get rolled back into the engine itself, or we implement a feature for a team that just makes their game look cool. From there we can write a blog post about the cool feature that we made. Sometimes we come in near the end of development or in the middle, so it's kind of hard to change processes of how the team's working because, at that point, they're just interested in getting the game shipped. For that reason, we find it better to work with teams at the beginning of development so we can run them through how to make a Unity game really efficient, and from there they should be good to go. We usually work with a couple teams from the beginning stages on long engagements that will probably take a year or more. At the same time, we also will help small teams with one-off things that take only a month or two. With these different teams, we provide a variety of assistance.

Advice to Kickstart a Career

The changes in development from early 3D to now are hard to describe. I think I saw the advent of the programmable shader pipeline[13], which I think completely changed everything. Then we saw the advent of compute shaders[14], which has made getting into 3D graphics way harder. Back in the day it was much easier, because all your work just consisted of polygons and lights, and that was it. Now there's all these features and techniques that people are using for specific things, and all these different ren-

13. Programmable **shader pipeline** allows the developer to customize some phases in the render pipeline (mostly the vertex processing phase and the fragment shader phase). It was introduced by OpenGL 3.2 in 2009.
14. A **compute shader** is a shader stage that is used entirely for computing arbitrary information. While it can do rendering, it is generally used for tasks not directly related to drawing triangles and pixels.

der paths[15]; it's gotten a lot more complex these days. Keeping up with all of these new additions and bringing them into my day-to-day work has been my biggest challenge.

I've been thinking a lot about how to help make things accessible for new graphics programmers. Early on, I think every programmer I knew had made a ray tracer[16], and now that's all the rage again — everyone's making ray tracers. I think that's a really good starting point, because it's just understanding the fundamentals of how light transport, reflections, refractions, and other essentials work. That's what helps graphics programmers build a solid foundation, and then they can build on that with more advanced skills. On top of that, definitely read every paper that's coming out and the latest things people are doing in the field. Quite frankly, there's just too much stuff to know.

I think every good graphics programmer out there has to be able to communicate with their artists. As a graphics engineer, you're responsible for getting them the tools they need to make sure they fit within performance budgets and memory budgets. You need to be willing to take criticism and understand their goals. One of the main pointers I can give is that when people just come up to you and ask for a new feature, oftentimes younger programmers will go off and immediately get to work on that feature. When they bring it back, though, they've made something that is only kind of like what the artist asked for. When the artist sees it, they ask for something different that will meet their goals, and start piling more stuff onto the programmer. So before you do anything else, it's good to have an understanding of what the problem your artists are trying to solve is. From there, you can get the requirements of what is needed to solve that problem, and work with them on how to present the feature to them. A lot of times programmers will make a feature and then put some "programmer UI" on it and say it's done, but it's completely unusable thanks to that UI. So figure out how to

15. **Render paths** are programs to affect the shading/rendering of lighting and shadow fidelity, along with other graphic details, with different performance characteristics.
16. In computer graphics, **ray tracing** is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects.

make it usable for people who are not you — that's also another key thing.

Further down your career when you might be managing, balancing that with development is a common problem, one that I still have to this day. I try to manage it by keeping the team small. When I was at EA, the team was made up of three to four other graphics engineers, so I was still able to do my usual work while managing the team. I don't think I was a terribly good manager at that time, but I was still able to keep a 50/50 balance. I think all my management jobs have been like that, where I try to keep the teams small and still be able to work while managing. However, if I know there's a task that's going to take months of my time or requires me to sit down and really concentrate on something, I'm not gonna have time. I have to pass something like that on to somebody else who can focus 100% on that. One task like that specifically was all of the vision modes in *Splinter Cell: Blacklist*, like infrared. While it's a cool task, it also requires working really closely with artists, and then the design of it is gonna change constantly. Because I didn't have the time, I handed it off to another guy on the team so I don't even have to think or worry about it.

A Lost Art

Engine programming is a lost art. It's important there are students and developers out there trying something like making a game engine, because it's just not being taught anymore. Then thinking about it from the perspective of wanting to make a game, there's the question of if one should spend two to three years making an engine for the game, or if they should just use Unity or Unreal and call it a day.

While there are some real monetary advantages to not writing your own engine, there's also different advantages to writing your own engine. You have to balance the pros and cons of the engine development process. There's some people online on Twitter who will scream that you have to write your own engine and you have to know it yourself, but I don't think that's the solution for everybody. At the same time, I think understanding

how engines work and the low-level stuff is incredibly important for all programmers. I think we'll get to a point where only a few people know how to make engines really well; we might already be at that point. *These days, I think hiring a graphics or engine programmer is close to impossible.* It's too hard to find people that know this stuff.

Interview conducted on October 15, 2018.

About the Team

Jared Ettinger is a creative writer and producer from New York. He is excited about the intersection of art and technology, particularly in video games and animation. He worked to further his production skills by keeping the rest of the team steady on the wild ride of building an engine.

Caleb Biasco started programming games in the Video Game Development Club at the University of Minnesota, and hasn't stopped since! Seriously. The madman is making game engines now, someone stop him.

Jacob Wilson is always taking things apart to understand how they work and sometimes they actually fit back together. His interests include tool development and long walks on the beach.

Chaojie Zhu has a background in software engineering from Shanghai Jiao Tong University, and has specific interests in game AI, self-driving vehicles and software engineering.

Yidi Zhu is a gameplay programmer/designer who enjoys making meaningful and playful interactive experiences. He is probably having too much fun in this swamp of game engine development.

About the Publisher

The ETC Press was founded in 2005 under the direction of Dr. Drew Davidson, the Director of Carnegie Mellon University's Entertainment Technology Center (ETC), as an open access, digital-first publishing house.

What does all that mean?

The ETC Press publishes three types of work: peer-reviewed work (research-based books, textbooks, academic journals, conference proceedings), general audience work (trade nonfiction, singles, Well Played singles), and research and white papers.

The common tie for all of these is a focus on issues related to entertainment technologies as they are applied across a variety of fields.

Our authors come from a range of backgrounds. Some are traditional academics. Some are practitioners. And some work in between. What ties them all together is their ability to write about the impact of emerging technologies and its significance in society.

To distinguish our books, the ETC Press has five imprints:

- **ETC Press:** our traditional academic and peer-reviewed publications;
- **ETC Press: Single:** our short "why it matters" books that are roughly 8,000-25,000 words;
- **ETC Press: Signature:** our special projects, trade books, and other curated works that exemplify the best work being done;

- **ETC Press: Report:** our white papers and reports produced by practitioners or academic researchers working in conjunction with partners; and
- **ETC Press: Student:** our work with undergraduate and graduate students

In keeping with that mission, the ETC Press uses emerging technologies to design all of our books and Lulu, an on-demand publisher, to distribute our e-books and print books through all the major retail chains, such as Amazon, Barnes & Noble, Kobo, and Apple, and we work with The Game Crafter to produce tabletop games.

We don't carry an inventory ourselves. Instead, each print book is created when somebody buys a copy.

Since the ETC Press is an open-access publisher, every book, journal, and proceeding is available as a free download. We're most interested in the sharing and spreading of ideas. We also have an agreement with the Association for Computing Machinery (ACM) to list ETC Press publications in the ACM Digital Library.

Authors retain ownership of their intellectual property. We release all of our books, journals, and proceedings under one of two Creative Commons licenses:

- **Attribution-NoDerivativeWorks-NonCommercial:** This license allows for published works to remain intact, but versions can be created; or
- **Attribution-NonCommercial-ShareAlike:** This license allows for authors to retain editorial control of their creations while also encouraging readers to collaboratively rewrite content.

This is definitely an experiment in the notion of publishing, and we invite people to participate. We are exploring what it means to "publish" across multiple media and multiple versions. We believe this is the future of publication, bridging virtual and physical media with fluid versions of publications as well as

enabling the creative blurring of what constitutes reading and writing.

Acknowledgements

The Isetta team would like to firstly thank all of the professionals who provided their time, knowledge, and advice through these interviews. What isn't seen from the text in the book is the additional time emailing back and forth trying to figure out a time that works in their crazy schedules, or the after hours editing we asked each of them to do. This book would have not been possible without them, nor would we as students learned or grown quite as much as we have, and for that we are incredibly grateful. Special thanks go to Cort Stratton, who has been a huge proponent of the project and wrote the poignant foreword even though he was experiencing the California wildfires.

A big thanks to Brad King, the editor and director of Carnegie Mellon University's ETC Press. He has provided us with countless hours of guidance and help in organizing, conducting, editing, and publishing these interviews — so basically the whole process!

Our faculty advisors on the project, Ruth Comley and Mike Christel, have also provided the team with thoughtful guidance and support throughout the length of the project. We would like to thank them for being patient with us and trusting us in completing the project, even if we gave them reasons to doubt.

Before we had our bearings in the subject matter of engine programming to even start asking directed questions, we had other professionals helping us who we'd like to thank. Alice Ching from Funomena gave us great advice on code reviews and really put things into perspective for us. Amit Patel and Rob Shillingsburg spent over an hour of their time with us at GDC, helping us flesh out our project idea and giving us great general advice on

starting a journey of developing engines and trying to help educate others, so we extend our thanks to them both as well. We would also like to thank Jason Gregory in part for creating such a great resource for us to use in learning engine development, as well as providing us confidence and reassurance that our project idea had value early on. Thanks to Walt Destler, ETC alumni, who spoke with us while we were developing our pitch to give us advice on engine development, especially as it relates to an indie game. And also thanks to Oliver Franzke who provided us with guidance in what to develop with our first game engine and stopping us before we made some poor decisions early on.

We have also been supported by our other ETC faculty, and special thanks to Dave Culyba for championing our project and helping shape it from its infancy, as well as Heather Kelley who helped connect us to industry professionals when we struggling to get our start.

Thanks to Yui Wei Tan who, although swamped with other coursework, volunteered to help a team of non-artistic programmers come up with amazing cover art.