# A Lost Art



headshot

***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.*

(The following is the edited transcription of a conversation we had with Raymond Graham.)

## 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 NBA2K, *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 little to no game development background 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]](#footnote-24) 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 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[[2]](#footnote-25) 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[[3]](#footnote-26), and with that if you wanted to get the most out of your graphics platform, you would have VU1 basically doing the draw calls [[4]](#footnote-27), batching[[5]](#footnote-28) up the 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[[6]](#footnote-29) in and you double buffer[[7]](#footnote-30), 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[[8]](#footnote-32), 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 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[[9]](#footnote-33); 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[[10]](#footnote-34) 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 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[[11]](#footnote-37) and Vulkan[[12]](#footnote-38). 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[[13]](#footnote-39) 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.

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[[14]](#footnote-41), which I think completely changed everything. Then we saw the advent of compute shaders[[15]](#footnote-42), 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 render paths[[16]](#footnote-43); 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[[17]](#footnote-44), 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 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 October 15, 2018.*

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. [↑](#footnote-ref-24)
2. 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. [↑](#footnote-ref-25)
3. **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. [↑](#footnote-ref-26)
4. 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. [↑](#footnote-ref-27)
5. 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. [↑](#footnote-ref-28)
6. **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. [↑](#footnote-ref-29)
7. **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. [↑](#footnote-ref-30)
8. **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. [↑](#footnote-ref-32)
9. **Virtual memory** is a memory management technique that abstracts uniformed memory space from different kind of storage device. [↑](#footnote-ref-33)
10. The Unreal Material Editor is a node-based graph interface that enables you to create shaders. For more see the [Unreal Documentation](https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/Editor). [↑](#footnote-ref-34)
11. **Metal** is a low-level, low-overhead hardware-accelerated 3D graphic and compute shader application programming interface (API) developed by Apple Inc. [↑](#footnote-ref-37)
12. **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. [↑](#footnote-ref-38)
13. **Entity-Component-System (ECS)** is an architectural pattern that follows composition over inheritance principle and is mostly used in games. [↑](#footnote-ref-39)
14. **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. [↑](#footnote-ref-41)
15. 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. [↑](#footnote-ref-42)
16. **Render paths** are programs to affect the shading/rendering of lighting and shadow fidelity, along with other graphic details, with different performance characteristics. [↑](#footnote-ref-43)
17. 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. [↑](#footnote-ref-44)