# Thinking about the Data



headshot

***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.

(The following is the edited transcription of a conversation we had with Martin Middleton.)

## 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 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]](#footnote-23) 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](https://twitter.com/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 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]](#footnote-26) 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 this engine you can use versus what really needs to be sidetracked. 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]](#footnote-27) 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]](#footnote-29) 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]](#footnote-30) and I can just give them a “simple transform”, which uses a Vector3 for the position and the Quaternion for 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]](#footnote-33) 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]](#footnote-34) and addressable with ID’s. 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]](#footnote-35), 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. 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]](#footnote-36) 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]](#footnote-38) 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’s a whole lot of rooms but there’s only a couple people in each room, and they’re all stranded from each other. To solve this, I ran a room defragmentation[[11]](#footnote-39) 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-number protocol[[12]](#footnote-41) that doesn’t really change so that it’s backwards-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 start with what specifically you want to learn (i.e. how application context[[13]](#footnote-44) 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]](#footnote-45). 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](https://isetta.io/interviews/CaseyMuratori-interview/) “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]](#footnote-47), or graphics wrappers where you just write forward to the wrapper and it handles the underlying OpenGL[[16]](#footnote-48) or DirectX[[17]](#footnote-49). There are also lighter frameworks that bring in bits and pieces like [meshoptimizer](https://github.com/zeux/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.*

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. [↑](#footnote-ref-23)
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. [↑](#footnote-ref-26)
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. [↑](#footnote-ref-27)
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. [↑](#footnote-ref-29)
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. [↑](#footnote-ref-30)
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. [↑](#footnote-ref-33)
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. [↑](#footnote-ref-34)
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. [↑](#footnote-ref-35)
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. [↑](#footnote-ref-36)
10. A **hash** is a structure that maps keys to values through a formula defined to convert structures into a index, typically the formula is constructed to avoid collisions between similar objects. The hash of the same object will always return the same value. [↑](#footnote-ref-38)
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. [↑](#footnote-ref-39)
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**. [↑](#footnote-ref-41)
13. **Application context** is the context, the set of data required to interrupt and continue a task, of an application. [↑](#footnote-ref-44)
14. An **audio renderer** is a system which plays/outputs spatialized sound, sound that is positioned in the world. [↑](#footnote-ref-45)
15. Simple DirectMedia Layer (**SDL)** is a hardware abstraction layer for audio, input, and graphics across multiple platforms.  
     [↑](#footnote-ref-47)
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. [↑](#footnote-ref-48)
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. [↑](#footnote-ref-49)