# NEEDS TITLE

headshot

headshot

***BIO***

(The following is the edited transcription of a conversation we had with Shanee Nishry.)

## Who is Shanee?

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 is 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]](#footnote-24). 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 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]](#footnote-26) 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, 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 more tight, 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 API’s 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[^2], 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 API’s 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?

[^2] **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.

The trouble often comes when your system tries to do too 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[[3]](#footnote-29), 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-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[[4]](#footnote-30) 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[[5]](#footnote-32). So at 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 expertise. 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[[6]](#footnote-34) division with rendering, and that way it’s really easy to discard a 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[[7]](#footnote-35) for 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[[8]](#footnote-37) to serialize stuff. I find that it’s really useful if 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 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[[9]](#footnote-40), and they are actively developing a new entity component system. With all that, we can see that even the big engines have to 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[[10]](#footnote-41) 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 quick. 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.*

1. **Binary size** refers to the size of the binary files built from source code and other assets. [↑](#footnote-ref-24)
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. [↑](#footnote-ref-26)
3. 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. [↑](#footnote-ref-29)
4. 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. [↑](#footnote-ref-30)
5. **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 detailed explanation, visit <http://alumni.media.mit.edu/~jorkin/goap.html> [↑](#footnote-ref-32)
6. **Quadtree** is a special type of tree data structure used in spatial partitioning. It recursively divide 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.* [↑](#footnote-ref-34)
7. **CDLOD** is short for the [paper](https://github.com/fstrugar/CDLOD/blob/master/cdlod_paper_latest.pdf) titled Continuous Distance-Dependent Level of Detail Rendering Heightmaps. It describes a technique for GPU-based rendering of heightmap terrains. [↑](#footnote-ref-35)
8. **JSON** (JavaScript Object Notation) is a lightweight data-interchange format 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. [↑](#footnote-ref-37)
9. 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. [↑](#footnote-ref-40)
10. 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. [↑](#footnote-ref-41)