# The Definition and Beginning of a Game Engine



headshot

***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.*

(The following is the edited transcription of a conversation we had with Adam Serdar.)

## 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]](#footnote-23) system as well as tool development with ScriptableObjects[[2]](#footnote-24). 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 import 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 of 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 framework, 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 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[[3]](#footnote-28) where all of the players come into a giant huddle to make the pile-ups even more realistic.

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[[4]](#footnote-29) 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[[5]](#footnote-31). 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 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[[6]](#footnote-33), the scriptable render pipeline[[7]](#footnote-34), 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[[8]](#footnote-36) 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 and OpenGL will have very specific calls that you’re basically 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[[9]](#footnote-38) 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 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[[10]](#footnote-39) 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[[11]](#footnote-41) 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. 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 graphic 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[[12]](#footnote-44) 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.*

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. [↑](#footnote-ref-23)
2. ScriptableObjects are scripts which cannot be attached to GameObjects but still store (serialize) user data. [↑](#footnote-ref-24)
3. An animation-blended system can be graph of multiple animations and transitions from an animation to another, i.e., idle to walk, and the blend system is how the animations are “mixed” together, it extrapolates from the starting animation to the ending animation. [↑](#footnote-ref-28)
4. 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. [↑](#footnote-ref-29)
5. Waterfall schedules 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. [↑](#footnote-ref-31)
6. 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 a shader is edited through nodes. [↑](#footnote-ref-33)
7. A scriptable render pipeline is a term specific to Unity, however can use in more generic sense of being able to customize the graphics pipeline for different platform requirements by downgrading/upgrading the level of detail, shader information, etc. [↑](#footnote-ref-34)
8. An audio library used for games, although it contains the word open it actually isn’t open-sourced. Its open-source counterpart is OpenALSoft. [↑](#footnote-ref-36)
9. 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. [↑](#footnote-ref-38)
10. Core render loop is the loop where the rendering function is called. The way the rendering occurs/is called varies from engine to engine, but is usually performed at the end of the main game loop. [↑](#footnote-ref-39)
11. NVIDIA PhysX is a proprietary realtime physics engine SDK created by NVIDIA. It is used in most commercial game engines such as Unity, Unreal, and Lumberyard. [↑](#footnote-ref-41)
12. Unreal’s Blueprint Visual Scripting System is the node-based scripting in the Unreal Engine used for gameplay scripting. [↑](#footnote-ref-44)