Entity-Component Systems & Data Oriented Design In Unity

Recently, there has been a lot of confusion in the Unity community about the “new way of programming“ in Unity.

I feel that most of the tutorials / articles relating to Unity’s implementation of an ECS Model are not doing a fair job in shedding light upon this topic.
So, if you are still not sure how or why to use the ECS model, this is for you.

Note: This article does not include the Job System at the moment.

Contents

Some Input On Current Game Dev Practices

First of all, the ideas presented in this article are not new. This is not news. For example, under the hood of game engines, these principles have been applied for decades.

Especially systems like rendering, physics & collision detection heavily rely on a more data oriented approach (this does not mean they are written in ECS, but they apply similar principles!). If we were to write these systems in the OOP / Entity-Component(EC) way, we would notice a significant decrease in performance.

On top of all of that, the world still looks a bit different. Especially small studios & Indie developers tend to keep their codebases entirely in an OOP / EC fashion and they try to enforce strict rules on how to write code. *May god have mercy with you, if you were to expose a field publicly instead of using a property!*

A few years ago this wasn’t a huge problem. However, with the increasing demand for bigger and better, more detailed as well as more immersive worlds, times are changing.

When walking around in the reconstruction of paris in Assassin's Creed Unity, right when the french revolution is happening, the player expects to be surrounded by a huge crowd of AI characters, where each of those characters is simulating complex crowd pathfinding & human behaviour.

Modern CPUs are extremly powerful, but you can still get to their limits quite fast, if you are actively designing your code against what the CPU prefers to work on.

With the ECS, Unity tries to free it's developers / users from those restraints and it tries to drift towards a more liberal way of writing code, that is about the data and ***not*** about your mental model of how the code should look like.

Also, note that the new ECS system is not at all supposed to completly replace the current EC Model (at least it ***should*** not, but more on that later). Some things are going to work better the **"**old" way, some things are going to work better with the new ECS.

The goal of all of this, is to solve your *current* problem according to your *current* needs.

Feel free to mix your code. You don't need a strict pattern for all of your code. All it does, is to satisfy *your* need for a uniform solution. But guess what: The best solution to your current problem is not the best solution for your next problem.

So, to cut a long story short: let's have a look at what ECS / DOD does different.

The Principles Behind ECS

So far, we have thought of our GameObjects as independent entities, that solve every problem they face on their own.
If you want to move a GameObject, you would throw a component on it, and move this single GameObject in the Update method. If you have a hundred of those, you would probably move a hundred GameObjects independently & call the Update method a hundred times.

The idea of an ECS, however, is to combine those GameObjects & handle their behaviour all at once with a single **system**.

Let's take a more detailed look at the title of this architectural pattern:

-**Entity**

Entities are the new GameObjects. The whole point of an entity is to associate data with it, they simply act as identificators (IDs). GameObjects have a lot of information (Name, instance ID, status of activity (enabled, disabled), layer, tag etc. and it is mandatory to have a transform component!), making them a heavy identificator.
Entities, on the other hand, are supposed to be really light weight. They don't have to

actively live in a scene (i.e. they don't require you to add a transform component), they are merely IDs that have no data on them on their own.

-**Component**

Components contain all of the data (e.g. a transform component could contain the position, rotation & scale of an entity.) They do **not** contain any behaviour.

-**System**

Systems handle all of the behaviour, and they do it at once for **all** of the entities asscociated with this system.

So, why **can** this actually be better ?

1. As you might have guessed now from the entity description, it may save quite a bit of memory. You have a lot more power over what data is actually on that entity, minimizing data garbage.
2. We are actually avoiding a lot of overhead that the current EC Model is introducing. There's a cost to calling the Update method (or whatever method) on a hundred GameObjects.
   If we call the Update method once in our system, and update all hundred GameObjects inside of that in a single batch, we have effectively cut this cost of.
3. It is a lot easier to parallelize our code. We can put different systems on different threads, and can even parallelize a single system easier, since we don't have to manage a bunch of GameObjects, but a single system & try to minimize data dependencies.
4. We prefer value types over reference types. Garbage Collector (GC) allocations & deallocations are avoided as much as possible.
5. We can optimize our code very easy to be more cache coherent.
   Since we are handling behaviour for a bunch of entities at once, we can lay out memory accordingly.

What's the downside of an ECS ?

1. As you can see, it is all about combining a bunch of entities. If we want to handle a single entity such as a player(at least singleplayer) for example, it becomes redundant to write such a system for it.
2. It's hard to debug specific entities. Everything revolves around the system, not the specific entities.
3. We usually have to write more code for the same functionality. (minus the pro's above)
4. It is not really taught to game programmers. If you have never done something outside of Unity / Unreal / [Insert popular engine here] you might as well think that the whole world is OOP.

With the introduction of the ECS, you, the programmer, will have a “new“ responsibility (not really new, as we figured out) that many devs have avoided for a long time:
You need to decide **when** to structure your code in **what** way.

Data Oriented Design (DoD)

We’ve learned a lot about the principles of ECS and Unity’s ECS implementation. So, what does data oriented design have to do with all of this ?

With the new ECS, Unity claims to switch from **object oriented** programming to a **data oriented** approach. By now, you will probably have heared the term DoD quite a bit, but what exactly does that mean ?

In contrast to OOP, where everything revolves around objects, DoD is all about the data you have. Instead of modelling our code the way we imagine & understand the world, we model our code according to the flow of data, to transform it from one state to another.

So, why would we want to do that ?

In the end, computers don’t work the way we do. When it comes down, all they can really work on is 0’s & 1’s, but they can do that really really good.
With that knowledge (which you probably already had before reading this) one should ask himself/herself why we are structuring code after our own thought process. Your CPU doesn’t understand the concept of an object. It doesn’t know about encapsulation and all that good OOP stuff.
So, when we write code this way, it comes with a cost: Your CPU is less efficient, simply because you’re not providing it with tasks it can actually perform really well.

If we want to be more efficient, we will have to let go of some of our habits, that are weighing us down. That doesn’t neccessarily mean to completly abandon OOP, but developers should stop to model everything according to their personal viewing of the world.
An important aspect of this idea is, that DoD is supposed to free us from Dogma. There is no benefit in strictly encapsulating everything (except for satisfying your desire to have a uniform structure.)
Often times, this dogma will make the code not even significantly slower, but it will also be harder to understand as we are creating obfuscating layers of abstraction.

When modeling our code according to our perspective of the world, we tend to create relations from the real world, even though they are fundamentally different sets of data.

For this, and a general overview on DoD, have a look at Mike Acton's awesome presentation on DoD!

Also, have a look at this presentation, which is about combining OOP & DoD.

I am no where near the experience Mike Acton has gathered about this topic, so here's his principles of DoD.

The Principles Of DOD As Layed Out By Mike Acton (Who now actually works at Unity!)

- The purpose of all programs, and all parts of those programs, is to transform data from one form to another.
- If you don't understand the data, you don't understand the problem.
- Conversely, understand the problem by understanding the data.
- Different problems require different solutions.
- If you have different data, you have a different problem.
- If you don't understand the cost of solving the problem, you don't understand the problem.
- If you don't understand the hardware, you can't reason about the cost of solving the problem.
- Everything is a data problem. Including usability, maintenance, debug-ability, etc. Everything.
- Solving problems you probably don't have, creates more problems you definitely do.
- Try solving the common case.
- **Reason must prevail.**

A Critical Comment On The ECS & Unity

Now that we have learned a bit about both, the ECS model as well as data oriented design, we can discuss the application of OOP, DoD, Unity's ECS and the claims Unity Technologies has been making over the last couple of months.

I have to say that I really like the way Unity is evolving and I love their work. Game developers slowly start to realize how powerful modern CPUs really are, and Unity is certainly being a big influence in that aspect.

However, there are a couple of things I am not too happy about.

- **The claim that Unity is now using a data oriented approach.**
  Unity is certainly drifting towards a ***more*** data oriented approach, but it is also

actively working against it in certain aspects. The whole idea of arranging your code in the ECS way is kind of anti-dod, because there should be no fixed solution, no fixed design pattern for all your solutions.
(I know that, on the other hand, Unity has pushed out Mike Acton's presentation "A Data Oriented Approach to Using Component Systems", so I think some of their claims are especially contradictory & confusing. However, I am also very well aware that a lot of this stuff is also very new to the Unity team.)

- **This includes the push to adapt to the new ECS.**
  I remember hearing Joachim Ante state in one of the presentations, that the team at Unity believes there should be one way of writing code, which I disagree with. (Remember Acton's words: "Different problems require different solutions").
- **OOP, at the moment, is the way game devs are taught to write games.**
  Wether this is good or bad is a different issue.
  So, I can also see that Unity is running themselves into a bunch of problems, as especially less experienced programmers will have a hard time understanding what's going on.
- **Unity is actively abstracting and obfuscating parts of the code for us.**
  The inject attribute is a perfect example for that. If you are an experienced programmer new to Unity, and you look at a ComponentSystem with a bunch of injected structures, you probably won't have the slightest idea what's going on. You don't see how your data is being injected into that system.
  How are those systems even called ? A default world is being generated for us in the background. Meaning, our ComponentSystems are going to execute anyways. If we don't want that, we have to actively stop that.

And what I probably dislike the most…

- **Unity, you don't even tell us, what you are doing behind the scenes!**
  For example, you are throwing around the Inject attribute in all those presentations & samples, but you don't explain how it works. The documentation is shallow at it's best at the moment. Go a little deeper when writing about core things like this in your documentation. It is really time consuming to step through the decompiled assemblies to finally understand what's going on. The programmer really should not have to do that.

With all that critique out now, I think it is also important to acknowledge the work Unity is doing here.
Unity is being pretty open about their changes, the engine is becoming more and more customizable, performance is greatly increasing, Unity is hiring experts in those areas and actively cooperating with it's users. I can see a lot of confusion in the community, but I can also see how Unity is taking action to liberate and revolutionize game programming.
So, even though I see tough times for Unity users to get adapted to these concepts, I also see the chance that game devs start to learn about more performance oriented approaches from early on.

The ECS is a great new addition, it runs really well and is going to give Unity users a lot more control and power (if they desire so).

I really like the way Unity is evolving and it is absolutely natural that with all those new features, it’s going to get a bit more messy.