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
- The Principles Behind ECS
- A Basic Game The "Normal" Way
- A Start Guide On Unity's ECS
    - Foreword
    - Content
    - The Core Of Unity's ECS (Documentation)
        - 1. Components
            - 1.1 ComponentWrapper
            - 1.2 ComponentDataArray
        - 2. Entities
            - 2.1 EntityArchetype
            - 2.2 EntityManager
        - 3. ComponentSystems
            - 3.1 Events
            - 3.2 Built-in Properties
            - 3.3 Injection
            - 3.4 EntityCommandBuffer
        - 4. World
        - Other
            - RunTimeInitializeOnLoadMethod Attribute

- Rewriting The Killerbird Game In ECS Style
  - 1. “Bootstrapping“
    - 1.1 Initialization
    - 1.2 “Bridging“
    - 1.3 Bootstrap Class
  - 2. The First ComponentSystem - KillerbirdSpawnSystem
    - 2.1 Dependencies & The Inject Attribute
    - 2.2 Adding Behaviour
    - 2.3 KillerbirdSpawnSystem Class
  - 3. KillerbirdMoveSystem
    - 3.1 Combining Data & Behaviour
    - 3.2 KillerbirdMoveSystem Class
  - 4. KillerbirdRemoveSystem
    - 4.1 EntityCommandBuffers
    - 4.2 KillerbirdRemoveSystem Class
  - 5. Side Notes
    - 5.1 Scale Component
  - Conclusion
- Data Oriented Design (DoD)
- A Critical Comment On The ECS & Unity

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.

A Basic Game The “Normal“ Way

Let’s have a look at a really basic game I’ve created for the purpose of this article.
Later, we’re going to rewrite this game in ECS style. Feel free to skip to the **A Start Guide On Unity’s ECS** section, if you like.

I am using 3 free asset packages from the Asset Store for the visuals.
Those are the packages

- https://assetstore.unity.com/packages/2d/characters/birds-pack-4-in-1-27774
- https://assetstore.unity.com/packages/2d/characters/tasty-characters-free-forest-pack-108878
- https://assetstore.unity.com/packages/2d/environments/2d-jungle-side-scrolling-platformer-pack-78506

The game principle is really simple. You will be able to control a player, that can move left and right. There are going to be a bunch of killer birds spawning in the Sky, trying to hit the player. Birds die (i.e. they simply get destroyed) when colliding with something.

The player controller. Nothing special. We’re just reading input and moving the player accordingly.

Player.cs

Furthermore, we have a Bird component that controls data (movement sped, rotation speed etc.) & behaviour (rotating, flying towards the player, diying on collision)

Bird.cs

Lastly, we have a simple Manager class that is responsible for spawning our killer birds in the sky.

BirdManager.cs

Note that this is just basic code for showcasing. I do not care about best practices etc. here. That is, we’re just doing basic stuff in Update, you won’t see stuff like [SerializeField] etc. just for the sake of having a light example.

Alright cool. Everything works. The game is not interesting at all, but it works fine as a little example. Now ask yourself, what could we do better ?

1. All of our birds move & rotate with the same speed. They have the same goal & the same behaviour. The only difference is going to be their position, and therefore their directional vector. So, with all that stuff in common, why would we let each bird operate on it's own ? This is a perfect example of a situation, that we can handle in a batch.
2. Instantiate as well as Destroy are slow functions, you can read up why that is all over the internet. But to give a quick explanation:
   They operate on random memory, may perform deep copying ((de)serializing objects) & let's not forget, they work on GameObjects. A pretty heavy identificator. The new ECS provides us with a lot faster methods for instantiating & destroying entities.
3. We can do a better job with alining memory for the CPU (i.e. laying out our memory more cache coherent)

Now that we have identified what we can do better, let's get started. From scratch.

A Start Guide On Unity's ECS

Foreword

First off, we're going to see that it is perfectly fine to write parts of your project with the ECS, and parts of it in the old fashion. The ECS is **not** superior to the EC Model in every aspect. (more on that later)

For this simple guide, I have oriented myself at the TwoStickShooter (Pure) sample Unity provides. However, I have changed quite a bit to demonstrate, that I don't fully agree with the application of the socalled "pure ECS" approach. The hybrid solution is pretty much just wrapping IComponentData types into classic MonoBehaviours, so that Unity can actually serialize that data. That's pretty trivial, so I won't get more in depth at that topic.

Apart from that, I have added comments all over the code. I found that the documentation on the samples Unity provided wasn't doing too good of a job to explain what is going on and especially **how** it is happening. So, I've went through the decompiled assemblies and tried to add a little description of how things like the Inject attribute work instead of just stating what they do.

Content

Therefore (and maybe because I'm a bit lazy), we're going to leave the player as it is. It works perfectly fine and is definitely not going to be our bottleneck.

However, we will rewrite the SpawnManager to actually be a Spawn **System** and we will rewrite the Killerbird MonoBehaviour to be a system that handles movement & rotation. There is also going to be a separate system that handles the destruction of our birds.

Unity has added a bunch of stuff in their samples, that show how debug information can be injected into the ECS. (E.g. in the pure EnemySpawnSystem the seed for the random number generator & stuff like that is recorded). I've left this out on purpose.
If you are interested in that nethertheless, you should be able to very well understand what's going on in Unity's samples after this guide.

Alright, let's get started.

The Core Of Unity's ECS (Documentation)

Before we start writing our game, we'll have to know how Unity's new ECS works in detail. In the following you'll find out about the core concepts of this ECS implementation. Feel free to use this as a reference point.

You might skip to the section Rewriting The KillerBird Game In ECS Style & come back here anytime to clarify.

Also, I am going to update this section more and more until Unity's documentation is more mature. Right now, for example, there's not much to read about the different events of a ComponentSystem. This is going to change in future updates of this documentation.

All the new types we are going to be using are shipped in new assemblies. So, make sure to include the respective namespaces.

Here are the new ones we're going to use

- Unity.Entities (Entity, EntityManager etc.)
- Unity.Mathematics (New math library)
- Unity.Transforms (Position, Rotation etc. there's a separate namespace for 2D, Unity.Transforms2D)
- Unity.Collections (NativeArray etc.)
- Unity.Rendering (MeshInstanceRenderer etc.)

1. Components

Components are nothing but data containers.

All we have to do to create a new component, is to create a new struct that implements the IComponentData interface.

```
public struct Foo : IComponentData { // data goes here }
```

Unity already provides a variety of components for you, such as Position, Rotation, MoveSpeed and various others.
With MonoBehaviours, we had a Transform component that would hold **all** transform data (position, rotation, scale, parent etc.)
Now, we want to be able to lay out our memory really efficient. Therefore, Unity is outsourcing all those properties into separate components.

Note, that ComponentData types are always value types, i.e. they are being copied by value. Since component data is not garbage collected, they should not contain references to managed objects.

1.1 ComponentWrappers

At this moment, IComponentData types cannot be serialized by the Unity editor.

We can wrap them into MonoBehaviours though, to access & alter our data directly in the editor.

```
[Serializable]
public struct Foo : IComponentData {}

public class FooComponent : ComponentDataWrapper<Foo> {}
```

Note, that we have marked `Foo` as serializable, which is of course required to display our struct in the editor. Also, Unity seems to denote a wrapped component by adding the suffix "Component" to the name.

However, we're not completly done yet.
The ECS requires a GameObjectEntity MonoBehaviour on the very same GameObject our wrapper lies on. This MonoBehaviour is simply grabbing all components in OnEnable , allowing our ComponentSystem's to access said components now.

All we have to do to make this work, is to throw a GameObjectEntity MonoBehaviour on our GameObject and it's simply going to work. Wrappers shipped by Unity such as the MeshInstanceRendererComponent will automatically add the GameObjectEntity MonoBehaviour for you.

1.2 ComponentDataArray

We'll be using ComponentDataArrays often for a collection of components.

ComponentDataArrays are NativeArrays. That is, they are neither allocated nor deallocated by the Garbage Collector. Memory is layed out linearly, instead of randomly as it is the case with normal arrays.

We can work with them just as we do with normal arrays. All we have to do, is to pass in the type of our component as the generic argument.

```
ComponentDataArray<Position> positionComponents;
```

2. Entities

Entities are merely Ids. They do not contain any data on their own. All we really use them for, is to associate a bunch of components with them. That also means that our entities don't need to live in a scene, which is the case for GameObjects.

If we look at the source code of the Entity structure, we can see, that this is literally all the data it contains

```
public struct Entity : IEquatable<Entity>
{
      public int Index;
      public int Version;

      // plus some operator overloads etc.
}
```

2.1 EntityArchetypes

By creating EntityArchetypes, we can define in advance what components we want an entity or a group of entities to be created with. This is not mandatory. However, it can speed up our system as Unity has the opportunity to arrange memory accordingly for us.

You could also see it as a prefab for entities. Although it does not save any component data for you, you can create one to function as a blueprint for your entities. So, for example, this is what the Archetype for a rolling ball could look like

```
EntityManager.CreateArchetype(typeof(Position), typeof(Rotation),
typeof(TransformMatrix), typeof(MoveSpeed), typeof(Ball),
typeof(MeshInstanceRenderer));
```

Instead of having to list all of the components we want when spawning a new Ball entity, all we have to do is to pass in our archetype.

2.2 EntityManager

The EntityManager is basically the link to our entities. We create and destroy entities using the EntityManager, we find entities or groups of entities through the manager and we set the data of components associated with specific entities through the manager. We can also add / remove components after the creation of entities through the manager.

Generally, if we're not writing a component system, we can get access to the EntityManager by calling

```
World.Active.GetOrCreateManager<EntityManager>();
```

All we have to do to create an entity, is to call the respective method and pass in either directly the types of components we want, or an EntityArchetype we have created in advance

```
EntityManager.CreateEntity(typeof(Foo), typeof(SomeOtherComponent),
…);
```

or

```
EntityManager.CreateEntity(fooArchetype);
```

We can also create entities in batches, by passing in a NativeArray. The CreateEntity method will take the length of the NativeArray and instantiate as many entities, which are then stored in the NativeArray.

```
var killerbirdSwarm = new NativeArray<Entity>(100, Allocator.Temp);
EntityManager.CreateEntity(fooArchetype, killerbirdSwarm);
```

We can set component data per entity

```
EntityManager.SetComponentData(myEntity, myComponent);
```

or add and remove (shared)components

```
EntityManager.AddComponent<Foo> (entity);
```

```
EntityManager.RemoveComponent<Foo>(entity);
```

3. ComponentSystems

The heart of the ECS model.

To create a Component System, all we have to do, is to create a class that derives from ComponentSystem!

```
public class FooSystem : ComponentSystem {}
```

Interestingly enough, Unity actually forces us to implement the OnUpdate function, which has always been optional with MonoBehaviours so far.

```
protected override void OnUpdate() {}
```

As you can see, when implementing the OnUpdate method, we are now overriding the base class's OnUpdate method (which is abstract). In contrast, MonoBehaviour events such as Update are grabbed through reflection at the very beginning.

Meaning, if you don't have Update implemented in your MonoBehaviour, it's not going to be called. On the other hand OnUpdate (and various other events) of a system will always be called. Even if they are empty.

3.1 Events

Apart from the OnUpdate method, which is mandatory for every ComponentSystem

```
protected override void OnUpdate() {}
```

we're provided with a couple more events that are being fired for us, similar to Awake, Start etc. of a MonoBehaviour.

```
protected override void OnCreateManager(int capacity) {}

protected override void OnDestroyManager() {}

protected override void OnStartRunning() {}

protected override void OnStopRunning() {}
```

3.2 Built-in Properties

Similar to MonoBehaviour's (e.g. gameObject, transform, etc.), the ComponentSystem comes with a bunch of predefined properties for us to use.

E.g. from inside a component system, there's no need to retrieve the EntityManager(see section **2.2** for more information). We cann access it directly through the EntityManager property.

```
public class FooSystem : ComponentSystem
{
    protected override void OnUpdate()
    {
        // No need to retrieve the EntityManager.
        // The system already provides a property.
        EntityManager.CreateEntity(...);
    }
}
```

There’s also a bunch of other stuff exposed for us, including the active world, component groups, PostUpdateCommand buffer etc.

3.3 Injection

A dependency on a ComponentSystem will prevent said system from operating, as long as the dependency is not satisfied (that is, OnUpdate etc. is not being called)

```
struct Data
{
    public ComponentDataArray<SpawnCooldown> spawnCooldown;
}
```

```
[Inject] private Data data;
```

In this example, spawnCooldown needs to be initialized with at least one SpawnCooldown component to satisfy the dependency.

```
public static void SetupComponentData(EntityManager entityManager)
{
     var stateEntity =
     entityManager.CreateEntity(typeof(SpawnCooldown));
     entityManager.SetComponentData(stateEntity, new SpawnCooldown {
     value = 0.0f });
}
```

Our newly created SpawnCooldown instance is then being injected into our data object.

Note: Even though we are only working with **one** `SpawnCooldown` instance, we are still using a `ComponentDataArray<T>`.
The inject attribute will look for a **generic type** (which `ComponentDataArray<T>` is, our component, on the other hand, is not.) It will then grab the first generic argument of the `ComponentDataArray<T>` type, that is `T`, which is in our case of type `SpawnCooldown`.
The type `SpawnCooldown` is then being injected as a dependency in our component system.

3.4 EntityCommandBuffer

Command buffers are commonly used in graphics programming. We can use them in Unity to avoid farious issues such as invalidated array’s.

EntityCommandBuffer’s are a way to record commands and execute them later.
The EntityCommandBuffer relys on an EntityManager. We can record most of the commands, that the EntityManager provides.

The ComponentSystem already provides a command buffer, called PostUpdateCommands.

```
private void SpawnAfterUpdate ()
{
    PostUpdateCommands.CreateEntity(...);
}
```

4. World

The world class pretty much fires up the ECS. It owns an EntityManager, so we can interact and manage our entities. It also owns our component systems.

The OnUpdate method as well as the other events the ComponentSystem class ships with, are being fired by the World class.

We can have different world instances, to handle different situations. By default, Unity creates a single instance at the very beginning of your game, and associates all of our component systems with this world instance.

To create a new world, all we have to do, is to call the constructor

```
var world = new World("Rendering");
```

we'll have to pass in a string, but you can pass in whatever you want. That's just the name of the world, so we can better manage what world handles which situation (in this example, this world would handle everything associated with rendering.)

We can then create an EntityManager, and do all the stuff we've done before with the default world.

If we don't want to use Unity's default world, we might dispose it. It will of course be the active world at the very beginning.

```
World.Active.Dispose();
```

We can grab all worlds and then dispose whatever world we want

```
World.AllWorlds();
```

Or we can simply dispose all worlds by calling

```
World.DisposeAllWorlds();
```

Other

A collection of other functionality, that doesn't quite belong to the ECS but still ships with it and is used in conjunction with it.

RunTimeInitializeOnLoadMethod Attribute

Unity provides us with a new attribute that we can put on our methods from any class, which we can use to initialize and prepare our entities, components and component systems.

```
[RuntimeInitializeOnLoadMethod]
```

The attribute takes the type of initialization as parameter, so we can actually decide when we want this Init method to be called. There are 2 different types right now:

```
RuntimeInitializeLoadType.BeforeSceneLoad
```

And

```
RuntimeInitializeLoadType.AfterSceneLoad
```

Which do exactly what the name tells you.

Rewriting The KillerBird Game In ECS Style

Alright. Time to put our knowledge about Unity's ECS into action. Don't worry if you don't have a clear picture of the ECS yet. We are going to work through this game step by step.

Also, I have left out comments in the code to not bloat up this article even more. There are loads of comments in the complete files, which you can find on GitHub (I'm going to link them here from time to time)

1. Bootstrapping

Before we actually run our systems, the very first thing we might want to do is to perform some initialization, preparation and, what I call, "bridging". For this, we might want to create a class that's often called "Bootstrap", which is going to handle this kind of stuff for us.

```
public class Bootstrap {}
```

Nothing special here at all. You can mark that class as sealed, if you're really fancy.

1.1 Initialization

Depending on the scale of our game, there's going to be more or less stuff we need and / or want to initialize and prepare for our systems. For example, we might want to create a bunch of EntityArchetypes or fill data for our systems. A lot of this is going to be a one-time operation, so it's natural to keep this out of our systems.

Of course, no MonoBehaviour's (our Bootstrap class is usually not deriving from any Unity class) also means that we don't have events like Awake, Start etc. that are being fired for us to initialize our data. Instead, for this case, we can use the new

```
[RuntimeInitializeOnLoadMethod]
```

Attribute.

In this case, we want our Init method to be called before the scene is loaded.

```
public class Bootstrap
{
      [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.Before
      SceneLoad)]
    public static void Init() { }
}
```

For our KillerbirdSpawnSystem, we're going to prepare an EntityArchetype and we will grab the MeshInstanceRenderer component (the ECS version of the MeshRenderer), which we have wrapped in a MonoBehaviour, so we can alter the look of our Killerbird in the Editor. (Note: Right now, there's no SpriteInstanceRenderer component. That's the reason why we are using a mesh.)

We will also grab another MonoBehaviour from the scene called "GameSettings", which we can use to expose common data such as spawn cooldown to the Unity editor.

Before we can create the EntityArchetype, we will have to get access to the EntityManager. We can do this by calling `World.Active.GetOrCreateManager<EntityManager>()`

This is what our Archetype is going to look like

```
entityManager.CreateArchetype(typeof(Position), typeof(Rotation),
typeof(TransformMatrix), typeof(MoveSpeed), typeof(KillerBird),
typeof(MeshInstanceRenderer));
```

all we have to do, is pass in all the components we want this entity to have. The MeshInstanceRenderer relies on a TransformMatrix component (this is where the renderer get's position, rotation & scale of the mesh from). We won't have to perform any operations with that component.

With a call to `GetLookFromPrototype<T>()` we grab the MeshInstanceRendererComponent from our our Wrapper, that we have placed in our scene. (more on that in the next section)

Lastly, we will setup some data for our KillerbirdSpawnSystem by calling
`KillerbirdSpawnSystem.SetupComponentData(entityManager);`

More on that later.

1.2 Bridging

“Bridging“ is what I call the interaction between MonoBehaviour’s and the ECS.

ECS components can’t be serialized at the moment (no worries, that’s going to change in the future). That is, we can’t edit our component data in Unity’s editor (which is obviously one of the greatest things about Unity)

What we can do, however, is wrap our components to old-fashioned MonoBehaviours. (See **ComponentWrapper**) Unity has already done this for us with the MeshInstanceRenderer component.

All we do in the before mentioned `GetLookFromPrototype<T>()` method, is to find our GameObject in the scene (which holds our MeshInstanceRendererComponent, the wrapped version of MeshInstanceRenderer) and grab the MeshInstanceRenderer component from it.

```
public static MeshInstanceRenderer GetLookFromPrototype<T>() where T
: MonoBehaviour
{
    var prototype = Object.FindObjectOfType<T>();
    var result =
prototype.GetComponent<MeshInstanceRendererComponent>();
    return result.Value;
}
```

As mentioned before, I don’t believe that the ECS **should** replace the OOP model. Meaning, our bootstrap class is also going to be great to function as the bridge between MonoBehaviour‘s <-> Entities / Systems.

The player for example is an entity that you usually want to create once at the beginning of the game. So, if your player actually is an entity (remember, in this guide our player is a normal MonoBehaviour) you could very well do that in the Bootstrap class, instead of creating a separate component system for that.

1.3 Bootstrap Class

So, here is what our bootstrap class is looking like by now.

Bootstrap.cs

In the Init function, which is being called by the `[RuntimeInitializeOnLoadMethod]` attribute, we’re creating an EntityArchetype for our Killerbird entity.

Then, we call the

```
GetLookFromPrototype<T>()
```

method which grabs the MeshInstanceRenderer component from a GameObject living in our scene. As you can see, there's already a wrapper for the MeshInstanceRenderer, called MeshInstanceRendererComponent, that's simply exposing our component to the editor.

Lastly, we're setting up some data for our first Component System, which we are going to talk about later.

2. Our First ComponentSystem - KillerbirdSpawnSystem

Alright, now we're ready to start our first ComponentSystem.

All we have to do, is to derive from `ComponentSystem` and implement the OnUpdate method. (See **ComponentSystem**)

```
public class KillerbirdSpawnSystem : ComponentSystem
{
    protected override void OnUpdate () { }
}
```

2.1 Dependencies & The Inject Attribute

The Inject attribute is one of those things Unity is always showing in their code, but not really explaining.

A dependency in our component system means, that our system is not going to operate as long as the dependency is not satisfied. This means, that e.g. OnUpdate is simply not being called by Unity.

We will have a dependency on our KillerbirdSpawnSystem, let's have a closer look at it.

```
struct Data
{
     public ComponentDataArray<SpawnCooldown> spawnCooldown;
}
[Inject] private Data data;
```

This looks probably pretty familiar to you by now, from all the videos Unity has pushed out so far. What's actually happening here ?

We define a struct, that is simply going to hold some data for us, that we're going to work with. (The SpawnCooldown component in this case, which holds nothing but a float value that represents our cooldown timer.)

When we now put the Inject attribute on our data field, **it means that we do not want to run our system until we have initialized our data** (meaning we have actually data that we can work with now.)

So, that's what is happening in the SetupComponentData method, that we have called earlier from within our Init method in the Bootstrap class.

```
public static void SetupComponentData(EntityManager entityManager)
{
    var stateEntity =
    entityManager.CreateEntity(typeof(SpawnCooldown));

    entityManager.SetComponentData(stateEntity, new    SpawnCooldown {
    value = 0.0f });
}
```

We are creating a new entity with a SpawnCooldown component, and initialize that component, fulfilling our dependency. The component is then being injected into our Data struct. Now, after calling this method, Unity will actually start running the system.

(See **Injection**)

2.2 Adding Behaviour

Next step is to actually **do** something in our system. In this example, we're going to use the OnUpdate method, which works exactly the same as the Update method of a MonoBehaviour.

All we do here, is to see if we're ready to spawn a new killerbird (IsReady()) and then actually spawn said bird (Spawn). Note, that we're accessing our GameSettings MonoBehaviour to grab common data we've exposed to the editor.

Keep in mind, that we're now working with value types most of the time, let's have a quick look at the IsReady method

```
private bool IsReady ()
{
     var time = data.spawnCooldown[0].value -Time.deltaTime;
     var ready = time <= 0.0f;

     if (ready)
         time = Bootstrap.gameSettings.spawnCooldown;

     data.spawnCooldown[0] = new SpawnCooldown { value = time};
     return ready;
}
```

Nothing exiting going on here, except of how we work with our SpawnCooldown component. Unity hasn't yet implemented ref return, meaning that at the end, when altering a value type, we will have to reassign the new value to our component.

In our Spawn method, we're finally creating our Killerbird entity and setting a bunch of component data for it. (see **Entities** & **EntityManager**)
Note how we literally **compose** our entity with giving it a position, rotation, look (renderer) etc. step after step.

2.3 KillerbirdSpawnSystem Class

That's all there is to the KillerbirdSpawnSystem. If you run the project now, it's going to work. There's nothing happening apart from Birds being instantiated, but it's working.

Here's the complete class

KillerbirdSpawnSystem.cs

3. KillerbirdMoveSystem

By now, we already know quite a bit about component systems. I assume you have understood the concepts so far, so I'm not going to explain the whole system again.

This system is going to handle the movement and rotation for **all** of our killerbirds. In The EC version of this game, we're moving every bird independently (which is pretty common in OOP)

We will access the Position, Rotation, MoveSpeed as well as the RotationSpeed component. Let's create a structure to hold that data and inject it into our system.

```
struct Data
{
    public readonly int Length;
    public ComponentDataArray<Position> positions;
    public ComponentDataArray<Rotation> rotations;

     public readonly ComponentDataArray<MoveSpeed> moveSpeeds;
     public readonly ComponentDataArray<RotationSpeed> rotSpeeds;
}
```

```
[Inject] private Data data;
```

The Length integer is also being auto populated for us (it's simply going to hold the size of our ComponentDataArrays, which are all of the same size)

Note, that we won't need an Init method this time, to initialize our components. That's because our Spawn method is already doing that for us. It's initializing every new spawned killerbird with a position, rotation etc.

In the OnUpdate method, we're going to iterate over all of our killerbird entities, and calling the Move method for each one of those.

```
protected override void OnUpdate()
{
    var deltaTime = Time.deltaTime;
    for (int i = 0; i < data.Length; i++)
    {
        Move(i, deltaTime);
    }
}
```

3.1 Combining Data & Behaviour

So far, we haven't worked much with our components. Now, we will actually interact with those and alter the behaviour of our killerbird entities by doing so.

All we have to do, is to read our data (position, rotation, …), change that and write it back into our ComponentDataArray's. Simple as that.

```
private void Move (int idx, float deltaTime)
{
    var resultRot = quaternion.identity;

    var targetPos = (float3)Bootstrap.player.position;
    var pos = data.positions[idx].Value;

    var dir = targetPos - pos;

    if (!dir.Equals(Vector3.zero))
    {
        var angle = math.atan2(dir.y, dir.x);
        var targetRot = quaternion.axisAngle(Vector3.forward,
angle);

       resultRot = math.slerp(data.rotations[idx].Value, targetRot,
       deltaTime * data.rotSpeeds[idx].value);
    }

     pos += math.normalize(dir) * data.moveSpeeds[idx].speed *
     deltaTime;

    data.rotations[idx] = new Rotation(resultRot);
    data.positions[idx] = new Position(pos);
}
```

Couple of things here.

- First of all, behind the scenes, Unity is already doing some work for us when using the Position & Rotation components. Changing position and rotation isn't enough, Unity is going to „translate" this data into a transform matrix (which is happening in the TransformSystem the ECS already ships with) and throws this into different systems such as a rendering system (MeshInstanceRendererSystem) and later physics, collision systems etc.

  So, Unity is not only pushing components for us, but it is also providing ComponentSystems that directly work with those components. All we have to do, is to read and write to those components, as done above.

- As you can see, Quaternion, Vector3 etc now have ECS counterparts (quaternion, float3, …). We can use those interchangeably though, as they contain explicit as well as implicit conversion operators to each other.
- There's a new math class to work with those new types. You won't find methods such as quaternion.axisAngle in the quaternion structure. You'll have to look into the math class.
- There's no physics component / system yet. That is, we can't move our entities by a rigidbody. For the moment, we'll have to alter the position directly. But since there's no collision system yet, that's not too big of a deal.
- We're working with ***value*** types, so we'll have to write our changed position & rotation back into our components.

3.2 KillerbirdMoveSystem Class

That's all we need to move and rotate our killerbirds. Inject our data into the system, iterate over aour entities, manipulate position and rotation, write back into components.

The code of the complete system

KillerbirdMoveSystem.cs

4. KillerbirdRemoveSystem

All there is left to do, is to destroy our killerbirds when they are "dead". In this "game", dead means they collided with something. Since there's no physics / collision system, we need to get a bit more creative right now. For this, we'll be doing a very basic bounds check. For each bird, we'll simply check if it's y position is close enough to the player's y position. Meaning, we're pretty much colliding with the player. If it is, the bird is dead and we can remove it.

As always, starting with the data that our system depends on

```
struct Data
{
    public readonly int Length;
    public EntityArray killerbirds;
    public ComponentDataArray<Position> positions;
}


[Inject] private Data data;
```

We will only need the position for the bounds check. Then all there is left, is to gain access to the entity, so that we can actually destroy it.

EntityArray is a NativeArray wrapped for us to work specifically with entities.

When adding a readonly integer called Length to our data structure, Unity will assume that this is the length of our array(s), so this is also being auto populated for us. Have a readonly integer that's not named Length in that strucutre and Unity is going to throw a bunch of exceptions at you.

Again, in the OnUpdate method, our actual logic is happening. We'll iterate over all our birds and see if they are out of bounds (= collide with the player, technically), which is happening in the IsReadyForRemoval method.

```
protected override void OnUpdate()
{
    for (int i = 0; i < data.Length; i++)
    {
        var ready = IsReadyForRemoval(i);
        if (ready)
            Remove(i);
    }
}


private bool IsReadyForRemoval (int idx)
{
     var height = math.abs(Bootstrap.player.position.y -
     data.positions[idx].Value.y);
     return height <= Bootstrap.gameSettings.boundsThreshold;
}
```

If it is, we'll destroy it in the Remove method.

4.1 EntityCommandBuffers

However, since we’re possibly altering our NativeArray’s (removing entities) as well as accessing them in the very same frame, Unity is again going to throw an exception at us.

InvalidOperationException: The NativeArray has been deallocated, it is not allowed to access it.

When removing an element, memory has to be de- and reallocated. So, accessing it when that is happening is a pretty bad idea. Unity’s solution to problems like this are **command buffers**.

Command buffers are especially common in graphics programming, and they do what the name tells you, they record commands. Those commands can then be executed later.

The ComponentSystem class already contains a EntityCommandBuffer called PostUpdateCommands, that we can directly access. It is called, as the name suggests, as soon as OnUpdate finished. We can do pretty much the same stuff with it, as we can with the EntityManager. Create entities, destroy entities, add components, whatever.

In fact, we’re using the EntityManager. It’s simply all being delayed for us.

So, let’s record a DestroyEntity command, to remove an entity after we have iterated over all of our entities (Post OnUpdate)

```
private void Remove (int idx)
{
  PostUpdateCommands.DestroyEntity(data.killerbirds[idx]);
}
```

4.2 KillerbirdRemoveSystem Class

To recap the system: We grab our entities and position components, we iterate over said entities, do a bounds check using the position of each entity and if the bounds check returns that the bird is technically dead, we record a command to destroy this entity.

The complete system

KillerbirdRemoveSystem.cs

And that’s all there is to do, to realize this little, not so funny game using the ECS.

5. Side Notes

A bunch of good-to-know things, that you don’t really need but I still want to include in this start guide. (I probably have already forgotten a lot of those side notes, so this section might get updated over time as well)

5.1 Scale Component

So, we have a Position and a Rotation component, but there's one more property to describe a GameObject's / Entitie's transform: The scale!

Right now, there's no scale component. The only way to scale your mesh, is to actually directly alter the transformation matrix. There's one problem though:
You can't do that in conjunction with the Position and Rotation component.

As mentioned before, under the hood, there's a ComponentSystem called TransformSystem operating on the Position and Rotation component. It's pretty much just grabbing those 2 components and throwing them together into a 4x4 matrix. However, when it does that, it completly overrides your transform matrix. Meaning:

- You change your matrix
- The TransformSystem overrides your changes & makes it therefore worthless.

The only way to avoid that right now, is to completly avoid the Position and Rotation component, and alter all 3 properties directly in the transform matrix.

Our Spawn method rewritten to directly use the transform matrix

```
private void Spawn ()
{
    var spawnCenter = new float3(0, 14, 0);
    var x = Random.Range(-15, 15);
    var pos = new float3(spawnCenter.x + x, spawnCenter.y,
spawnCenter.z);

     var entity =
     EntityManager.CreateEntity(Bootstrap.killerbirdArchetype);
     EntityManager.SetComponentData(entity, new MoveSpeed { speed =
     Bootstrap.gameSettings.moveSpeed });
     EntityManager.SetComponentData(entity, new RotationSpeed { value
     = Bootstrap.gameSettings.rotationSpeed });
     EntityManager.SetSharedComponentData(entity,
     Bootstrap.killerbirdRenderer);

    // Create a new matrix with a rotation and position
    var transform = new float4x4(quaternion.identity, pos);
    // Create a scale matrix with the factor of 100
    var scale = float4x4.scale(100);
    // Multiply both matrices
    var result = Unity.Mathematics.math.mul(transform, scale);
    // Write our transform matrix into our component
     EntityManager.SetComponentData(entity, new TransformMatrix {
     Value = result });

}
```

- At first we create a 4x4 matrix (float4x4) with a position and rotation (don’t ask me why there’s no constructor to include a scale, I’ve not the slightest idea.)
- Then, we create a scale matrix. **Note, that this is not scaling your matrix, it creates an empty matrix that has the passed in scale!**
- Next, we multiply both matrices. There’s no multiplication operator, so we will have to use the mul function, that you can find in the new math class.
- Lastly, write our matrix into the TransformMatrix component, that e.g. the MeshInstanceRendererSystem is going to work on.

Conclusion

If we ignore all the comments in the code, and the fact that we are confronted with many new principles as well as a whole new framework, it really isn’t that much more code. Unity is doing a lot of work for us in the background.

So, what benefits did we now get from rewriting our game to be a mix of OOP (Player) and ECS (Killerbird- Spawn, Move, RemoveSystem) ?

- Multithreading is **easy.** We barely have any dependencies on our entities. The only thing a killerbird depends on, is it’s own position, rotation & the player’s position. Especially when using the new Job System, we can easily put all of that stuff onto different threads.
- We have a clear separation of data & logic. You want to know what data we are working with ? Look at the components. You want to know how this data flows ? Look at the systems. That’s it.
- We have arranged our memory **way** more efficient for the CPU. When Moving & rotating a killerbird, all we throw in is the position, rotation & speed.
  When we use MonoBehaviours & their Transform component, we have to load in the **whole** object. We can’t just take the position separately & work with that. The whole object is going to be loaded into memory.
- We are relying heavily on value types, instead of reference types. Meaning, we avoid Garbage Collector (GC) allocations and deallocations.
- As a result of all of the above, performance improved significantly! Just by arranging our code & memory differently.
  On my machine (Intel i5-6600) those are my results
    - Classic scene: 2.3 – 3.0 milliseconds per frame (330-430 fps)
    - ECS scene: 1.6 – 1.9 milliseconds per frame (625-525 fps)

In this really basic game, performance improved for us by about 200 fps.

Of course, the jump in performance will depend on your specific game and how you apply the principles we have talked about in this article.

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 acknowlodge 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.