The Core Of Unity’s Entity Component System (Documentation)

Contents

Foreword

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.

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 EntityArchetype

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.