A simple guide to get started with Unity ECS

Marios Koutroumpas Follow
Apr 5 · 5 min read

How to easily implement the Entity Component System pattern in your Unity DOTS project.

Photo by Ricardo Gomez Angel on Unsplash

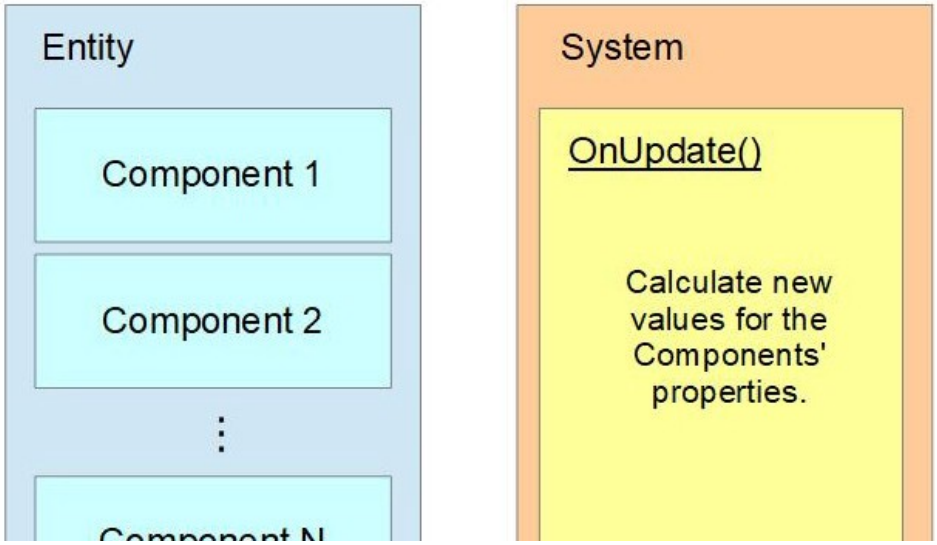
Introduction

Before we start creating our first game using Unity DOTS, it is a good idea to present a few simple ECS coding practices in a way that makes it easier to learn and utilize. As we have already discussed, the ECS pattern involves a novel approach of writing code in Unity, focusing on data and their flow between entities, instead of routinely extending standard GameObject classes with properties and methods. This results in many advantages, such as cleaner code and a more efficient development cycle. To that end, in the following paragraphs, we walk through the code of a revised implementation of the "Cubes Demo" sample app demonstrated in the previous article (see reference [4]).

The ECS pattern

Instead of working with standard MonoBehaviours, the new methodology involves Entities, Components, and Systems. So instead of creating subclasses of MonoBehaviour containing both the data (fields) and the operations (methods) performed on them, these two artifacts now become completely separated. As a result, Entities become representations of GameObjects mapped to a single identification key.

Components are typically implemented using C# structs. Several Components can be attached to a specific Entity and represent its various in-game characteristics; for example "speed", "weight", "damage taken", etc. Finally, Systems consist of implementations that impose a specific behaviour on every Entity that matches their preset criteria. Typically, those criteria involve filters matching the existence (or absence) of specific Components in a specific Entity.



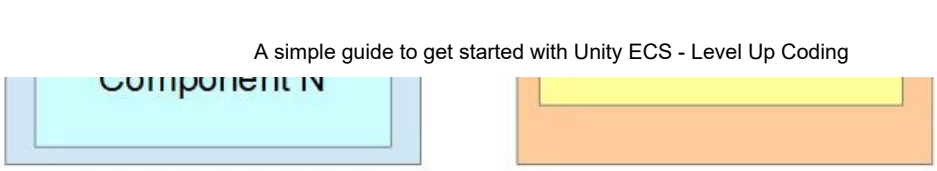


Image 1: Illustration of the ECS concept.

Summarizing all the above, we conclude that when writing ECS game code, there is no need to think of our GameObjects in terms of specific roles anymore. Concepts such as “Player”, “Health Pack”, “Enemy”, “Weapon”, etc. become only abstract, linguistic representations of our game entities, so we can now focus solely on the game’s data (i.e. Components) and the operations or interactions (i.e. Systems) that we wish to apply on them. This approach has a positive impact on the reusability and maintainability of our code, as well as on its development speed. But its most important advantage is that it allows for a massive increase in execution performance through the use of Unity’s Job System (see reference [5]) and Burst Compiler (see reference [6]). In the following paragraph we discuss the coding details of this concept.

Unity ECS implementation

It is quite easy to become familiar with the implementation of the ECS pattern in Unity, by keeping the following simple guidelines in mind:

1. Implement ***Unity.Entities.IConvertGameObjectToEntity*** to define your Entities and then convert your prefabs to Entities by using ***Unity.Entities.GameObjectConversionUtility***. Use ***Unity.Entities.EntityManager*** on your newly created Entities, in order to attach your Components to them.

2. Write Components by simply implementing the ***Unity.Entites.IComponentData*** interface. That’s all that is needed to create your custom data structures.

3. Implement your Systems by extending the ***Unity.Entities.JobComponentSystem*** class (a subclass of Unity.Entities.ComponentSystem). This allows you to utilize Unity’s Job System, which manages parallel execution of multiple threads in your application.

In the following section, we can see how these suggestions are implemented in our revised sample “Cubes Demo” project (source code in reference [3]).

Firstly, we define the Cube class by extending MonoBehaviour and implementing the IConvertGameObjectToEntity interface.

```
using Unity.Entities;
using UnityEngine;

public class Cube : MonoBehaviour, IConvertGameObjectToEntity
{
    public void Convert(Entity entity, EntityManager entityManager, GameObjectConversionSystem c
    {
        entityManager.AddComponent(entity, typeof(CubeComponent));
        entityManager.AddComponentData(entity, new MoveSpeedComponentData { Value = Random.Range
    }
}
```

Cube.cs hosted with ♡ by GitHub view raw

Snippet 1: Defining the Cube class

Then, we instantiate our Cubes by converting our cube prefab to an Entity.

```
public class URPTestTwo : MonoBehaviour
{
    private EntityManager _entityManager;

    [SerializeField]
    public GameObject Prefab;
    [SerializeField]
    public int NumberOfObjects = 1000;
    [SerializeField]
    public int XSpread = 10;
    [SerializeField]
    public int YSpread = 10;
    [SerializeField]
    public int ZSpread = 10;

    private void Start()
    {
        var settings = GameObjectConversionSettings.FromWorld(World.DefaultGameObjectInjectionWo
        _entityManager = World.DefaultGameObjectInjectionWorld.EntityManager;
        var entity = GameObjectConversionUtility.ConvertGameObjectHierarchy(Prefab, settings);

        for (var i = 0; i < NumberOfObjects; i++)
```

```
        {
            var position = new Vector3(
                Random.Range(-XSpread, XSpread),
                Random.Range(-YSpread, YSpread),
                Random.Range(-ZSpread, ZSpread));

            var entityInstance = _entityManager.Instantiate(entity);

            _entityManager.SetComponentData(entityInstance, new Translation { Value = position });
        }
    }
}
```

InstantiateEntities.cs hosted with ♡ by GitHub view raw

Snippet 2: Instantiating Cube Entities from prefabs

To represent the individual speed of each Cube instance, we implement a MoveSpeedComponentData component class that implements IComponentData and will be used in our custom System. Also, we create an additional component class named CubeComponent, that will be used within our System implementation as a filtering criterion for Entities.

```
using System;
using Unity.Entities;

[Serializable]
public struct CubeComponent : IComponentData { }

[Serializable]
public struct MoveSpeedComponentData : IComponentData
{
    public float Value;
}

```

ExampleOfTwoComponents.cs hosted with ♡ by GitHub view raw

Snippet 3: Implementing two custom Components

We can now add the Cube script shown in Snippet 1 to our Cube prefab, in order to have our custom Components (shown in Snippet 3) automatically added to every new cube

that will be created and spawned in the scene.

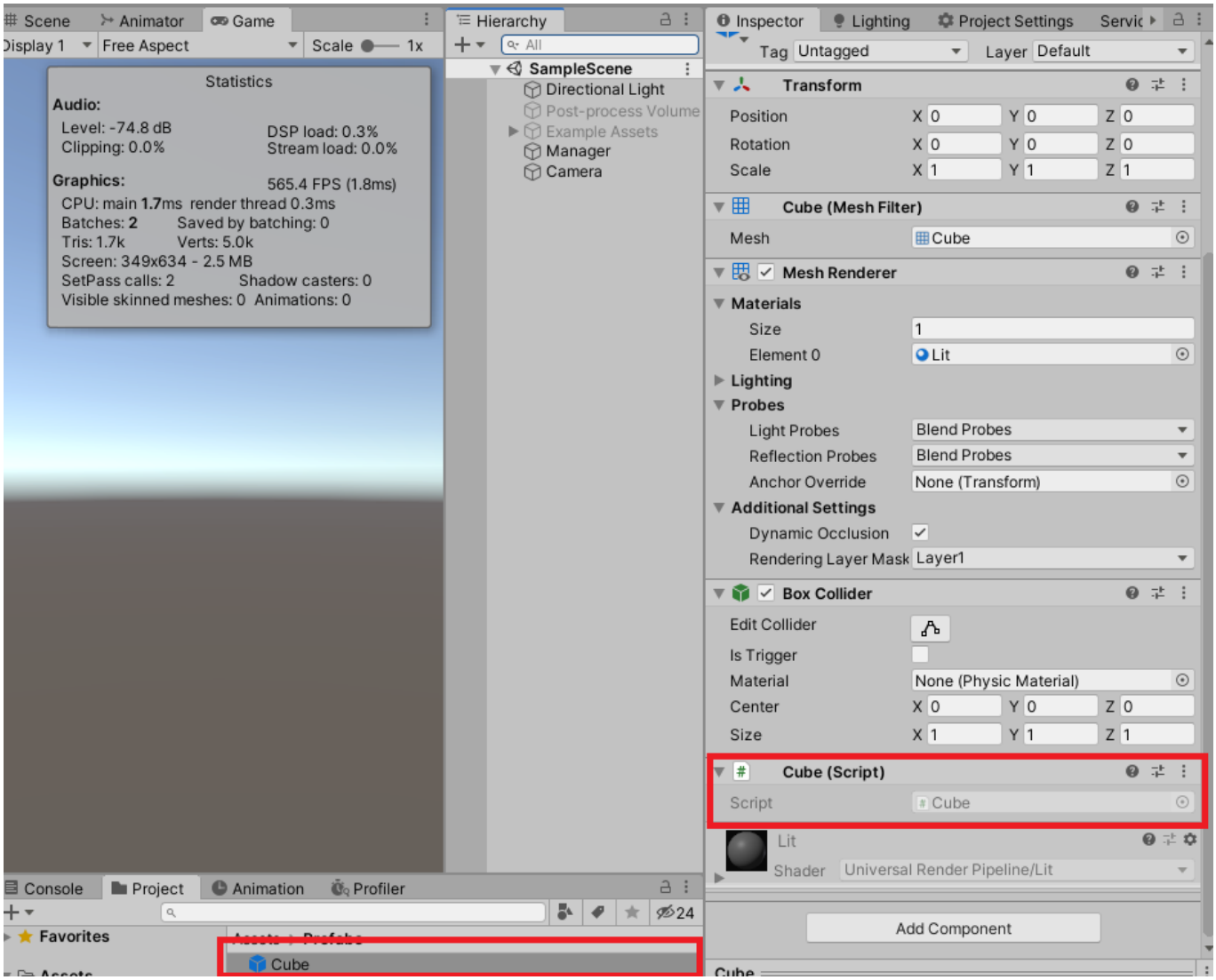


Image 2: Adding the Cube script to the Cube prefab

Finally, we create a System class (ZTranslationSystem), which includes an inner Job struct (TranslateJob) and implements the OnUpdate() method of the JobCompomentSystem abstract class. In the OnUpdate() implementation, we only have to schedule the execution of a TranslateJob instance. In the Execute() method of the TranslateJob struct, we implement our actual cube movement algorithm; in the case of this demo, a plain linear style of motion. We have also added a dummy “heavy task” method, in order to simulate a much more complicated and demanding pathfinding algorithm, which would definitely be present in an actual, fully-featured commercial game.

```
1   public class ZTranslationSystem : JobComponentSystem
```

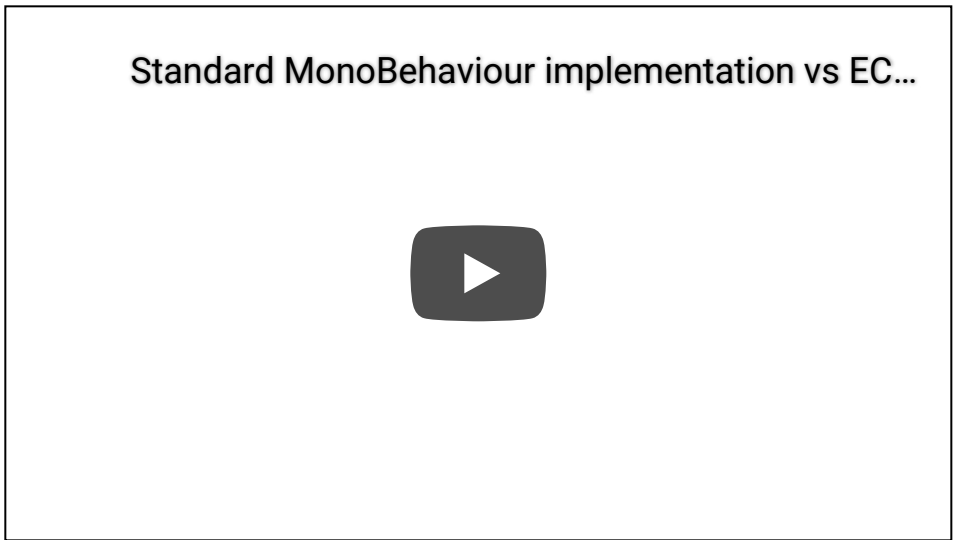
```
{
    [BurstCompile]
    [RequireComponentTag(typeof(CubeComponent))]
    struct TranslateJob : IJobForEach<Translation, MoveSpeedComponentData>
    {
        [ReadOnly]
        public float DeltaTime;

        [ReadOnly]
        public bool MinusDirection;

        public void Execute(ref Translation translation, ref MoveSpeedComponentData moveSpeed)
        {
            translation.Value += new float3(0f, 0f, (MinusDirection ? -1 : 1) * moveSpeed.Value
            Helpers.AddDummyHeavyTask();
        }
    }

    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        var job = new TranslateJob
        {
            MinusDirection = true,
            DeltaTime = Time.DeltaTime
        };

        return job.Schedule(this, inputDeps);
    }
}

public static class Helpers
{
    public static void AddDummyHeavyTask(int length = 1000)
    {
        float value = 0f;
        for (var i = 0; i < length; i++)
        {
            value = math.exp10(math.sqrt(value));
        }
    }
}
```

Snippet 4: Extending the JobComponentSystem class to create a custom System

The Execute() method accepts two different component types as parameters (Translation and MoveSpeedComponentData) and the TranslateJob class is decorated with two attributes: the first one, RequireComponentTag(CubeComponent), is responsible for filtering all Entities which contain a component of type CubeComponent. The second attribute, BurstCompile, instructs Unity to compile this class by using the Burst Compiler.

Please note that Systems in general (i.e. all classes that inherit from Unity.Entities.ComponentSystem) will always run automatically, without the need to be instantiated or called from a different part of the project. This means that the OnUpdate() method is invoked on every frame, just as is the case with the classic MonoBehaviour.Update() [see reference 1] method.

The runtime result is shown in the following video. A frame rate of more than 200 fps is easily achieved on an average PC, providing smooth motion to one thousand Cube instances, despite the heavy calculation task attached to each one.



Video 1: The final result

Conclusion

Implementing of the ECS pattern in Unity might seem somewhat complicated in the beginning, but once a developer familiarizes themselves with the three simple guidelines presented above and actively applies them to create a few apps on his/her own, it quickly becomes a powerful tool for writing quality code and creating highly performant applications.

References

1. MonoBehaviour.Update() method documentation (Unity3D)
2. Unite Copenhagen 2019 — Converting your game to DOTS
3. “Cubes Demo” source code
4. High — performance VR on a low-end Android phone
5. Unity Job System documentation
6. Unity Burst Compiler documentation

Unity3d Game Development Csharp Coding