A Start-Guide On Unity's Entity Component System

Contents

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.

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 Of This Guide

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**, rewrite the Killerbird MonoBehaviour to be a system that handles movement & rotation and write a new system to handle the destruction of killerbirds.

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.

If something is unclear, have a look at the documentation file in this repository (The Core Of Unity's ECS)

Alright, let's get started.

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.

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 (or better, about 0.7 – 1.1 milliseconds per frame).

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.