Unity DOTS - Data Oriented Technology Stack

Contents

[Introduction 2](#_Toc175584081)

[Setup 3](#_Toc175584082)

[C# Burst Compiler 3](#_Toc175584083)

[ECS – Entity, Component, System 4](#_Toc175584084)

[Entities and Components 4](#_Toc175584085)

[Custom Scripts and Baking 6](#_Toc175584086)

[Systems 7](#_Toc175584087)

[C# Job System 7](#_Toc175584088)

[Analysis of DOTS in Games 9](#_Toc175584089)

[Examples from the Industry 9](#_Toc175584090)

[Application in my Demo 10](#_Toc175584091)

[Final Thoughts 11](#_Toc175584092)

[References 11](#_Toc175584093)

# Introduction

Video Games are often held back by one limiting factor: performance. Many developers desire to create sprawling environments with countless assets - creating extensive, immersive experiences. This directly contrasts with many player’s hardware capabilities. Not everybody has the budget to afford the latest graphics cards, but for performance-intensive games, powerful hardware is a must.

Of course, there are some solutions to this problem, like incorporating different quality settings into a game. This gives the player the option to reduce the visual fidelity, thereby increasing the performance. However, this approach does come at the expense of reduced aesthetics and only helps to a certain degree. The optimal solution would be to have *performance by default*.

This is exactly what *Unity DOTS* promises. DOTS stands for *Data Oriented Technology Stack*. It is comprised of three main systems that together have the ability to increase the performance of an application dramatically.

The first element of DOTS is the *ECS* or *Entity Component System*. Instead of Unity's usual *gameobjects*, it uses *entities*. These entities have *components* that hold data - and *systems* that apply logic to that data. This approach clearly separates between logic and data, optimizing the way the data is stored which makes it more efficient to be handled by the CPU.

Secondly, there is the *C# Job System*. It takes advantage of multi-core processors by making the application *multi-threaded,* meaning that multiple different calculations can run in parallel instead of one after the other, resulting in a massive performance boost.

The last component of DOTS is the *C# Burst Compiler*. It takes code written by the developer and compiles it into highly optimized *machine code*. This works especially well for code that uses the C# Job System.

In the following article, I will explain how to set up and start to work with the Unity DOTS System, I will analyze examples from the industry that make use of the system, and lastly, I will show how I went from this:

A red circle with white text

Description automatically generated

… to this:

A red oval with white text

Description automatically generated

# Setup

In order to set up a Unity project to use the DOTS system, a few packages need to be installed: the *Entities* package, the *Entities Graphics* package, the *Jobs* package, and the *Burst* package. Also, the entities graphics package does not work with Unity’s Build-In Render Pipeline, so either the *URP* or *HDRP* needs to be installed and set up beforehand. Lastly, the *Unity Physics* package together with its sample *Custom Physics Authoring* is required to use the physics system in combination with DOTS.

Additionally, to get the best performance, in the *Project Settings*, under *Editor*, the option *Enter Play Mode Options* should be checked, but the boxes *Reload Domain* and *Reload Scene* should remain disabled (Unity, 2024).

A screen shot of a black background

Description automatically generated

# C# Burst Compiler

As an introduction to the topic, this chapter will explain the C# burst compiler. It seems like the simplest element of DOTS to use but has big implications for the whole workflow. In general, it increases the performance of an application by compiling its code into highly optimized native code (Unity, Burst User Guide, 2023).

Easily enough, the burst compiler works automatically, without any input from the developer. All that needs to be done, is to put the keyword *[BurstCompile]* in front of any methods or structs that should make use of its functionality.

The burst compiler does come with a big limitation though: it only works on code that does not allow the use of managed types. While common primitive types like *bool*, *char*, *float*, or *int* are supported, this means that it does not work for *classes*, *lists*, or Unity components like *Transform*, *Rigidbody*, or *Colliders*. This is why the DOTS workflow is designed to mostly make use of unmanaged types. For every managed type, there are unmanaged alternatives that should preferably be used in the DOTS workflow to make use of the burst compiler's advantages. Note that these alternatives are no equivalents to the managed types and do not work as a replacement in all cases.

|  |  |
| --- | --- |
| **Managed Type** | **Unmanaged Type Alternatives** |
| Class | Struct |
| Vector3 | Float3 |
| Array | NativeArray |
| List | NativeList |
| Transform | LocalTransform |
| Colliders | Physics Collider |
| Rigidbody | Physics Velocity, Physics Mass, Physics Gravity Factor, Physics Dampening |

In the following chapter, the extensive implications of this limitation will be made clear.

# ECS – Entity, Component, System

## Entities and Components

Now we know that one goal of the DOTS workflow is to limit the use of managed types to make as much use of the C# burst compiler as possible. Another objective is the optimization of memory. In the normal Unity workflow, data is stored seemingly randomly in the memory. So when the CPU tries to access that data it needs to jump around a lot which is a performance-intensive task. On the other hand, the DOTS workflow is optimized to order data into specific groups of components which makes it much faster to read. This feat is achieved by the ECS which clearly separates between data and logic.

A screenshot of a video game

Description automatically generated

(CodeMonkey, 2024)

So how to write ECS code? First of all, to use entities instead of Unity’s usual gameobjects, a *subscene* must be generated inside the hierarchy. All objects that will be created or moved inside this subscene will automatically be turned into entities. This process is called *baking* and will result in two different representations of the same object: firstly as a gameobject with Unity’s usual components, and secondly, during runtime, the object and all of its components will be baked into an entity with entity components. To easily have an overview of both of these representations, a second *inspector* tab can be opened, and set to *runtime mode* instead of *authoring*. Inside the normal inspector, all components are shown as usual, whereas in the runtime inspector, the corresponding entity components are presented. As an example, I have created a simple cube:

*Gameobject inspector view (authoring): Entity inspector view (runtime):*

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

The developer can just interact with the gameobject side in the editor and most of Unity’s standard components like *transforms*, *mesh renderers*, etc, will be automatically baked into entity components by the engine.

There are some exceptions though. Physics components like Rigidbody and Collider do not get baked into entity components automatically. Instead, the Custom Physics Authoring sample inside the Unity Physics package contains some alternative gameobject components. Some examples are *Physics Shape* and *Physics Body*, which will be automatically baked into their corresponding entity components like *Physics Velocity*, *Physics Mass*, *Physics Collider*, etc. These alternative gameobject components and their respective entity components have similar fields to the standard ones which makes working with them very comparable to the normal Unity workflow.

For custom *Monobehavior* scripts, the baking needs to be done manually.

## Custom Scripts and Baking

So we understand entities, entity components, and the conversion of standard Unity gameobject into the new workflow. To convert Monobehaviors, some manual coding is required though. This will incorporate the previously mentioned unmanaged types. In parallel to the automatic baking of components, the manual baking of custom scripts also has a gameobject and an entity side. As an example, I will show how to bake a simple Monobehavior that holds a single float variable for the rotation speed of an object.

A black screen with white text

Description automatically generated

Here we can see the entity side of that component. Instead of the class Monobehavior, it inherits from the *IComponentData* interface, which means that it can not be attached to a gameobject. Additionally, to gain the maximum performance increase, the entity side should only make use of unmanaged types, so the component is a struct instead of a class and only contains unmanaged types like this float. Whenever we need a managed type on the gameobject side, an alternative unmanaged type must be used on the entity side. For some types like Vector3 and Float3, the conversion happens automatically, while others require some manual coding.

A computer screen shot of a program code

Description automatically generated

This is the system's Monobehavior side, which will be attached to a gameobject. The *Bake* method will then automatically be called to transfer the data into the entity component. So now we have a normal custom script component to work with in the Unity editor, that will then be automatically turned into an entity component.

*Gameobject inspector view (authoring): Entity inspector view (runtime):*

A screenshot of a computer

Description automatically generated A grey screen with white numbers

Description automatically generated

## Systems

So far we have created a standard cube gameobject with a custom script that holds a rotation value, which will be baked into an entity with an entity component. Since entity components only hold data, the logic will be implemented in another script called a system. There are two different types of systems. When the system absolutely needs to deal with managed types like gameobjects, transforms, delegates, etc. it should be a partial class that inherits from *SystemBase*. If the system only needs to deal with unmanaged types like ints, bools, structs, etc. it should be a partial struct that inherits from *ISystem*. The second option has much better performance because it can be burst compiled, so one should always try to use the ISystem variant unless they absolutely need to use managed types. An example of that would be if the system has to communicate to a gameobject through an event, which can only be used in a SystemBase.

Both variants can implement multiple methods of which the most used will be *OnCreate*, *OnDestroy,* and *OnUpdate*. They work similarly to the usual *Start*, *OnDestroy*, and *Update* of the standard workflow.

Now, these systems do not get attached to any objects, since they do not inherit from Monobehavior but they also do not have a gameobject variant that gets baked into an entity version. So how do you determine which entities the system should run on? There are multiple options, but a simple one is *Queries*.

A computer screen shot of a computer code

Description automatically generated

This example rotates the previously created cube. It is a system that makes use of the unmanaged structs *RotateSpeed* and *LocalTransform* and it uses the OnUpdate function that gets called every frame to call a query. Inside the query, a list of entity components is specified, so that the code will be executed for all entities that have those components. In this example, all entities that have localtransform and rotatespeed components will be rotated. Note that every component that is accessed must be marked with either *RO* or *RW*. These stand for read-only and read-write. They limit whether the component should be changeable or simply used for reading values. RO is faster and should be the default unless a value does need to be changed.

## C# Job System

At this point, we have all the tools we need to create a simple scene with the DOTS workflow. Gameobjects get automatically baked into entities with corresponding components. Custom scripts are manually baked. Logic and data are separated and the code is optimized by the burst compiler. These steps already grant a big performance optimization but there is one more part that takes this optimization one step further: the C# job system.

This system should be used whenever the same part of the code needs to run multiple times, like a pathfinding system on several characters that all walk simultaneously. Normally, Unity is a single-threaded application, meaning that it calculates the paths one after the other. Only when the path of one character is calculated, will Unity start to calculate the next one. In contrast, multi-threaded applications can run many calculations a once, limited by the number of cores a CPU has. This leads to a natural decrease in total calculation time.

A diagram of a pathfinding

Description automatically generated

(CodeMonkey, 2024)

A computer screen with text

Description automatically generated

This code does exactly the same as before, it rotates cubes, only now on multiple threads. The job is created as a partial struct inheriting from *IJobEntity*. The components that are required for this job to run, are specified in the *Execute* method instead of the query like before. We still need to specify whether a component will be used to read and write or only to write but in the context of the job we use *ref* instead of RW and *in* instead of RO. Notice, how the delta time variable needs to be passed into the job since it has no reference to *SystemAPI*. To run the job, multiple methods can be called, but to make it run multi-threaded, we use *ScheduleParallel*. Now we can rotate multiple cubes at once, instead of one after the other.

# Analysis of DOTS in Games

## Examples from the Industry

So now that we have a general understanding of what the specific components of DOTS achieve, how they do it, and how to use them, let us take a look at the industry and find some examples of how professionals utilize the workflow’s advantages.

One game that was a very early adopter of the DOTS workflow, starting with version 0.2 in 2019, is *V Rising* by *Stunlock Studios* (Rasmus Höök, Why source access to DOTS is a game changer | Unite 2022, 2022). It is a massively well-received, open-world, vampire game that can either be played in single- or multiplayer. The studio behind the game faced the challenge of wanting to create an environment that was bigger than anything they had ever done before. Additionally, they desired to make the game feel alive by containing a large number of characters that noticeably act on their own, without the player's influence. As if that was not enough, they also had some high expectations for the graphics, wanting to incorporate a lot of volumetric fog into the game for instance (Rasmus Höök, Making V Rising, 2022).

To keep performance high, despite the demanding requirements, the studio made use of the main DOTS packages: *Entities*, *Entities Graphics* (formally named *Hybrid Renderer*)*, Jobs*, *Burst,* and *Physics*, while using Unity’s *High Definition Render Pipeline*. When it comes to the specific features of DOTS, the studio especially paid attention to adding or removing as few components as possible, claiming to have improved the performance of the game by 10% using that philosophy alone (Rasmus Höök, Why source access to DOTS is a game changer | Unite 2022, 2022).

They also used ECS’s subscenes to divide the world map into over 200 subscenes of 160 meters by 160 meters of in-game size each. This division not only ensures only having to render a smaller fraction of the entire world map at any given time, but the optimized performance of the subscenes also enables quick loading. All entities that are not in any of the currently loaded subscenes still exist in memory but are disabled for saving performance. To still achieve the mentioned effects of a world that feels alive, when enemy characters are in a disabled subscene, they are represented by a dummy object that is still enabled. This dummy object has limited functionality but can still execute tasks such as walking around, so the player might notice some changes even in areas that are not currently loaded (Rasmus Höök, Making V Rising, 2022).

Another philosophy that the developer of V Rising applied, is the splitting of entity components. Before I mentioned the difference between managed and unmanaged types, as well as their corresponding system types SystemBase and ISystem. Since the unmanaged version ISystem has a higher performance, in part because unmanaged types do not need garbage collection, the developers decided to split entities that need both managed and unmanaged types into two. This makes it possible to use the unmanaged side of the entity without losing performance (Rasmus Höök, Making V Rising, 2022).

A screenshot of a computer screen

Description automatically generated

These are only some examples of how Stunlock Studios made use of DOTS advantages over the normal Unity workflow, but they give a hint at the depth end possibilities that the system offers. Overall, the developers of Stunlock stated to be very satisfied with DOTS performance, especially the simplicity of the burst compiler, the easy parallelization of the jobs system, and the separation between data and logic. Even though they encountered some bugs in the required packages, they claim to want to use DOTS in the future as well (Rasmus Höök, Making V Rising, 2022).

## Application in my Demo

Finally, let us take a look at how I implemented the DOTS system in my demo. The goal of this demo is to showcase the performance increase of the system over the standard workflow by making full use of its components.

In the first step, I created a demo scene that does not use DOTS as a reference. The scene includes a freely movable camera inside a giant box, as well as the ability to spawn and despawn spaceships, 1000 at a time. These spaceships get assigned a random position inside the box and a velocity in a random direction. They preserve 100% of their energy upon collision with the walls or other spaceships and their rotation is frozen. Instead, they always face the direction that they are heading toward. In the top left corner, the user can view the current FPS as well as the number of spaceships that are currently spawned in. With the standard Unity approach, it takes around 10.000 spaceships before the FPS drops to an average between 1 and 3.

Now for the DOTS demo, the functionality stayed the same, with the only difference of using DOTS behind the scenes. First of all, every entity is created in the same subscene. This makes the most sense since I want to render all entities at any given time anyway, so no need to load or unload them. Secondly, four different types of gameobjects will be baked into entities: the walls of the box that the spaceships will spawn in, the spaceships themselves, the player, and a game-manager that is used for spawning the spaceships. All of these gamobjects, except the game-manager, need components for physics calculations. For that, I used the mentioned replacement components for Unity’s Rigidbody and Colliders. These PhysicsBody and PhysicsShape gameobject components will automatically be baked into entity components, that will optimize the physics calculations.

I also created three different systems for these entities, all with their own custom entity component script that contains as much of the necessary data as possible. Firstly, there is a system for player movement that calls a query on all entities that have a LocalTransform, PhysicsVelocity, and PlayerMovementDOTS components, which is only the player. Secondly, the spaceship spawning system takes care of the creation of the spaceships. It calls a query on all entities that have the SpawnSpaceshipDOTS component, which is the game-manager. Finally, a system for the rotation of the spaceships makes use of the job system to schedule a function on multiple threads, that rotates the ships in the correct direction. Unfortunately, the first two of these systems need to deal with managed types, the first with the camera, and the second needs to pass an event to change the UI counter of spaceships on screen. However, the third system is based on ISystem, allowing it to make full use of DOTS features. I also only could use the job system in one place, since it is the only code that needs to run in parallel.

With these improvements, I managed to increase the number of rendered spaceships before the FPS drops to 1, from 10.000 to around 65.000, so an increase of approximately 65-fold.

# Final Thoughts

Working with and researching the DOTS system made two things abundantly clear to me: it is a giant leap in terms of performance but it is so complex that I only managed to scratch the surface of the system.

Having a performance increase of the mentioned magnitude is amazing. It can be all the difference of achieving to make a game that would otherwise not be possible. Every time there are a large number of objects that need to be rendered the performance impact is felt the most. This can be the case for RTS, tower defense, hack and slash, but also some open-world or platformer games. Additionally, the simplistic nature of the C# burst compiler can also be seen as a low-hanging fruit that could increase the performance of all games.

On the other hand, working with DOTS requires a lot of practice. Basically, a whole new way of writing code must be learned. There are new components that must be used instead of Unity’s standard once, at least for the physics calculation. There are new unmanaged data types that must be used instead of managed types. Even though these alternatives should in theory be similar to use, in practice there are many differences, which lead to a change in usage. Additionally, the splitting of code into data and logic might be optimized for performance, but it creates many more lines of code and takes longer to write. The job system is more difficult to debug, and many different exceptions and errors can occur.

Despite these challenges, I feel like I made good progress in learning how to use the system. I gained a general overview, that can act as a starting point for further strengthening my knowledge of DOTS, and the giant performance increase makes all the difficulties worthwhile.

# References

CodeMonkey. (2024). *Unity DOTS Explained (ECS, Job System, Burst Compiler - 2024 Still Updated)*. Retrieved from YouTube: https://www.youtube.com/watch?v=Z9-WkwdDoNY

Rasmus Höök, F. a. (2022). Making V Rising. (L. G. Hasan Al Salman and Director of DOTS and Multiplayer, Interviewer)

Rasmus Höök, F. a. (2022). Why source access to DOTS is a game changer | Unite 2022. (S. S. Sebastian Schöner, Interviewer)

Unity. (2023). *Burst User Guide*. Retrieved from Unity Manual: https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html

Unity. (2024). *Entities overview*. Retrieved from docs.unity3d.com: https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html