



Advisory report: Unity's Data Oriented Technology Stack

Author: **Tim Wolfram** (tim@rebels.io, timothy_wolfram@hotmail.com)

Company: **Rebels**

Version: **1.0**

Date: **07/07/2021**



Table of Contents

Introduction	4
Time	4
Structure	4
Subjective vs objective	4
Summary and advice	5
Summary	5
Advice	7
Official advice from Unity	7
Subjective or personal advice	10
1. Recommendation to Rebels' Unity Team	10
2. Recommendation to Rebels for a possible follow-up research	11
3. Advice based on situation	12
1. What is DOTS?	15
1.1. Burst and the DOTS Runtime	15
Intrinsics	16
Modding support	17
DOTS Runtime	17
Job System	17
Entities:	18
The Entity-Component-System pattern:	18
Data layout: Chunks, Archetypes and cache misses	18
Physics: Stateless physics	19
Features	20
Pros & cons	21
Jobs	21
Burst	21
Entities	22
2. How to implement DOTS	23
Installing the correct Unity editor version and packages	24
Step one: Install the Unity Editor	24
Step two: Set up your project and scene	25
Step three: Install DOTS packages	25
Step four: Settings for increased editor performance	25
Debug settings	25
Play mode settings	26
Implementing the Jobs package	26
Implementing the Burst package	27
Implementing the Entities package	28
Component data types	28
ComponentData	28



DynamicBuffer	29
SharedComponentData	30
Systems	30
Entity queries	30
.WithoutBurst, .Run, and .WithStructuralChanges	30
Entities.ForEach and Job.WithCode	31
Filtering the Entities.ForEach query	31
Using MonoBehaviours with Entities.ForEach	32
Entity interaction (data relationships):	32
ComponentDataFromEntity	33
EntityCommandBuffer (ECB)	33
GameObject to Entity Conversion	33
Authoring components	34
Finding up-to-date DOTS information	34
Building DOTS projects	35
3. Personal experience and opinions	36
Personal opinion about DOTS	36
User friendliness	36
Problems found during prototyping	37
Instability	37
Naming issues	38
ComponentDataFromEntity	38
Velocity vs PhysicsVelocity	39
Entities.WithName	39
Physics triggers, collisions with child objects, compound colliders	39
Issues with SubScenes	41
My opinion on "High Performance C#"	42
Broken sample projects and outdated information	43
Things I am currently satisfied with, or believe can't be improved upon	44
Writing ECS code	44
File structure	45
Things I feel need improvements or am unsatisfied with.	45
DOs and DON'Ts	47
Common issues	48
My entities are invisible or pink	48
The samples provided by Unity do not work	48
The physics are broken	49
A method I need is missing	49
I can't reference an entity in the inspector	49
Important sources	50
Sources from Unity Technologies	50
Community sources	50



Glossary	52
Source list	54
Attachment 1: DOTS Survey	59
About the respondents' level of experience, and their teams	60
Knowledge about DOTS concepts (Data-Oriented Design, Multithreading) before using DOTS	62
Package usage and usability	63
Final notes	68
Attachment 2: Prototype	71
Attachment 3: Benchmarks	72



Introduction

This document is an advisory report to Rebels' Unity Team. It contains information about Unity's new **Data Oriented Technology Stack (DOTS)**, written by Tim Wolfram during a graduate internship at Rebels.

Time

This advisory report was written between February and July of 2021. Any references to "currently", "now", et cetera, will refer to this period in time. Any references to DOTS packages will refer to packages released alongside Entities version 0.17 on the 22nd of January, 2021 unless explicitly stated otherwise (such as when talking about the 2021.1 compatible versions of Burst and Jobs)

Structure

The advisory report is structured as follows:

Following this introductory chapter is a short summary of this document and advice about DOTS. This consists of the official advice given to developers by Unity Technologies, followed by the advice I personally would give to Rebels' Unity Team. might change at some point in the future, it also includes my personal recommendation to developers based on their specific situation.

Chapter 1 is an explanation of DOTS and its concepts, pros/cons, et cetera.

Chapter 2 explains how to implement DOTS' most important packages: Burst, Jobs, and Entities.

In Chapter 3 I will share my opinion about DOTS and go into detail about the challenges I faced while writing the prototype. It shows how to solve some of these problems so that in the future, other developers won't have to waste time trying to figure out the root cause, or can at least eliminate a potential reason if the problem they're facing looks similar but requires a different solution.

The end of this document has some helpful resources that are often referred to in this document; there is a page dedicated to [important sources](#) such as Unity's manual pages, Unity Learn courses or helpful video tutorials. There is also a glossary of terms commonly used in this document.

Subjective vs objective

Much of the information listed will talk about the pros and cons of DOTS versus the old GameObject/MonoBehaviour structure. However, besides the measurable differences such as performance, the preference for DOTS or GameObjects can be very subjective. The first two chapters will mostly be objective information shared by Unity or the Unity community, and the last chapter (chapter three) will mostly be my subjective experience and opinion.



Summary and advice

The following document is an advisory report on the use of **DOTS (Data Oriented Technology Stack)** by Rebels' Unity Team. The information in this document applies to Unity version 2020 LTS and the DOTS packages released on or after January 22, 2021. In the case of the Entities package, for example, this will refer to version number 0.17.

This chapter will provide a brief summary of this document. It will also provide advice on the use of DOTS: First there is a short explanation about Unity Technologies' official advice. Then, the author will share his own opinion about whether a developer should or should not use DOTS, mostly specific to Rebels' Unity Team. Some of this advice is partially based on the results of a survey posted to Unity's DOTS forum, the results of which are attached to this document ([Attachment 1](#)).

This summary and conclusion/advice will contain no new information, quotes or links to new sources. Any advice given is based on information referenced in this document, and any conclusions are, unless specified otherwise, solely subjective statements by the author, Tim Wolfram, based on personal experience and research in the time spent working at Rebels for a graduate internship.

Summary

Unity's **Data Oriented Technology Stack (DOTS)** is a collection of packages mostly focused on making Unity perform significantly faster and giving developers a higher degree of flexibility. The core of DOTS consists of the packages **Jobs**, **Entities** and **Burst**, however, there are many more packages currently in development, which are all made to be compatible with these packages. Many important functions of Unity don't have a (usable) DOTS implementation yet, such as animation, sound, and user interface.

The Job System is a system that makes it easier to write multithreaded code in Unity by providing safety systems that prevent race conditions. It allows developers to write blocks of code that can be scheduled to be executed on another thread at some later time. The safety system does come with restrictions as you can only use **blittable data types**.

Burst is a compiler written for DOTS that compiles Jobs very efficiently. It works on a subset of C# which Unity Technologies refers to as "**High Performance C# (HPC#)**", which is a more restricted version of C# which omits some functionality that would slow down performance, such as allocation and garbage collection.

Implementing the Entities package allows users to use the **Entity-Component-System (ECS)** structure. When used with the principles of **Data-Oriented Design** in mind, this can increase performance by optimizing the amount of **cache misses**. ECS also has some advantages in readability and reusability of code.

These three packages are considered the “core” of DOTS, and when used together can enable incredible performance. To learn more about this, and the best practices when using DOTS, **it is highly recommended to follow the following courses on Unity Learn:**

- [“What is DOTS and why is it important”](#) is a more in-depth introduction of the basic DOTS concepts and why they matter.
- [“Entity Component System”](#), a collection of video tutorials on the basic concepts of ECS, Jobs and Burst.
- [“DOTS Best Practices”](#), a much more in-depth explanation of the concepts of DOTS and Data-Oriented Design, optimizing code to reduce the amount of cache misses, and how to correctly design DOTS code.

Reading Unity's manual page for the Entities package is also a good start when it comes to learning the basic concepts of ECS and DOTS syntax, but the amount of information might be overwhelming for someone who's new to these concepts, at least compared to some other tutorials like the courses on Unity Learn.

For those who prefer video tutorials, Unity's Youtube channel has some videos about the basics of DOTS. However, most of these are outdated as DOTS syntax has significantly changed since the first preview packages. It is recommended, when using Youtube, Google, or any other search engine, to use advanced search features to find recent tutorials to avoid getting tutorials with outdated information, as the packages (especially the syntax) are still undergoing heavy changes between DOTS version releases. “Turbo Makes Games” is one of the few Youtube channels mostly focusing on game development using DOTS, and has a lot of up-to-date tutorials about DOTS packages. For developers who prefer video tutorials over text, this is a great resource for beginner-friendly tutorials that are up-to-date with the latest packages of DOTS.

DOTS, in its current state, does not prioritize user friendliness. In fact, the lack of user friendliness is one of the major factors currently holding it back from being a “Release”/“Verified” package according to Unity's CTO. Stability can also be an issue, as DOTS is relatively new and most packages are still marked as “Experimental” or “Pre-Release”/“Preview”. Therefore, it is not the best tool for rapid prototyping. If time is of the essence, especially if your programmers are inexperienced with DOTS systems, sticking to only using Objects/MonoBehaviours is highly recommended, apart from Burst and Jobs. However, the possible benefits in performance and decreases in code complexity can make DOTS absolutely worth the trouble, if you're willing to get over the steep learning curve and are willing to potentially spend some extra time to fix the issues you could run into.

The DOTS team has shared their philosophy on what they believe the DOTS workflow will look like in the future: they want to make the experience roughly the same with Entities as with GameObjects. However, we are still far off from this experience as a lot is still missing, such as being able to select an Entity in the Scene View and edit its component values.



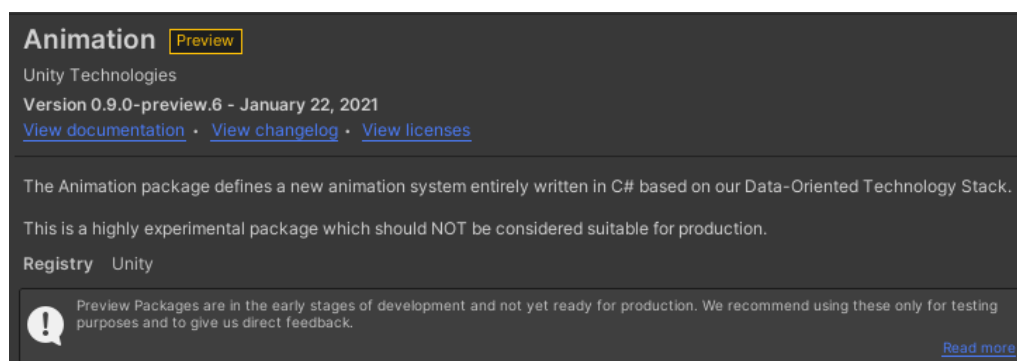
That is not to say that developing with DOTS quickly is impossible; an experienced programmer could, in some scenarios, match or even beat the speed of writing a similar MonoBehaviour implementation. However, in the current state of DOTS, a lot of boilerplate code is still necessary in a lot of cases. Despite this, ECS seems to be the unanimously preferred method of writing code by DOTS forum users.

Advice

Official advice from Unity

Any time you're installing a package that is not considered to be "production-ready", you will be warned in the description of this package. Unity separates packages in three categories:

- **Released** (called "Verified" in older versions) are packages that Unity considers to be stable and production-ready.
- **Pre-Release** (called "Preview" in older versions) are considered to be almost complete, but are not visible in the package manager by default. The visibility of these packages can be set in the preferences window.
- **Experimental packages** are not necessarily part of the official roadmap; as the name implies, they are experimental and therefore could be unstable, missing important features, or have little to no documentation. As seen in the package Unity Animation, experimental and preview packages will warn the user of their status prior to installing.



Unity, therefore, does not officially recommend any package that is not in the "Released" status. In fact, Unity actively discourages the use of most DOTS packages by hiding these packages from the Unity package manager, even when preview packages are enabled. A manual install is necessary for these packages, to prevent users who aren't fully aware of the experimental nature of these packages from attempting to use these systems before they're production ready.



The Unity Learn course "[What is DOTS and why is it important?](#)" has some insight about why or when to use DOTS, and why it would be a good idea to already start adopting DOTS:

DOTS is a new product and is still in its infancy. Because it is in continual development, you will see the API evolve and mature as we continue to work on making it the best it can be.

For now, you can think of DOTS as an extension to the Unity engine. DOTS is still in preview, so it is available for you to start exploring and using in your projects. You can create a new project from scratch using DOTS (and MonoBehaviors where required) or you can convert elements of your existing project to DOTS.

If you identify DOTS as an area that is going to be important to you, we recommend you start learning about DOTS as soon as you can. Although elements of the DOTS API will change, the fundamental principles of DoD and how to approach your projects using DOTS will stay the same.

This course explicitly defines two important reasons to use DOTS

- Improving performance, battery life, iteration and scalability
- Working with large amounts of data

It also mentions that DOTS might be very important in the near future, and that you will need to adopt it as soon as you can to stay competitive.

Unless you are immediately seeking performance improvements in the short or medium term, it can be hard to determine if or when to move to DOTS.

DOTS is highly likely to provide some level of performance improvement in almost every application. Areas include performance, battery life, iterating, and project scalability. You won't see any performance degradation by moving to DOTS, but assessing the cost of moving to DOTS is crucial, especially for projects when only small improvements are gained.

For all applications, DOTS is suitable for working with large amounts of data, such as open world environments or complex structures that use large amounts of the same materials. DOTS is also suitable for repetitive elements by sharing common data across instances to reduce memory accesses.

It's important to consider that DOTS is going to help you develop high-quality content in the future that a Unity without DOTS might struggle to deliver. For example, the standard games and Unity projects of today were the AAA games of the past. You will need to adopt DOTS to stay competitive in the future."

This course also contains some advice about the use of DOTS based on the context of your team or project:

For game indie devs and freelancers:

DOTS can help you offload some expensive operations in your game and help improve performance, especially for repetitive processes.

Many lightweight games, such as for mobile, do not come close to maximizing hardware performance. Even for those that do, this may not be a primary concern. However, as games continue to evolve and increase the demands of hardware, it is sensible to prepare for using DOTS in the future. Again, Project Tiny provides a solution for developing smaller apps and games using DOTS.

In cases where you may not have an immediate need to start working with DOTS, it's a great idea to get ahead of the curve and upskill yourself in DOTS so that you are ready for when DOTS is standard practice in Unity development.

For games studios:

DOTS in its current format can help you start to reach scale and performance you couldn't achieve before, in Unity or otherwise. In particular, extended battery life time and thermal control alongside the nature of code reusability afforded by DOTS are key benefits. Extended performance in these areas also allows you to develop for more low-end devices, especially outside Western markets, that have otherwise restrictive hardware limitations.

Having R&D groups begin to work in DOTS will help you begin to understand the best approaches you can take going forward, and will inform you of the current features and areas that will give the most performance benefits and development impact. DOTS does not aim to replace the role of engine teams, but frees up engineers to innovate in their own areas of expertise, such as shadows or shaders.

Subjective or personal advice

This chapter contains advice for Rebels' Unity Team based on personal opinions and experience. Therefore, before I can give any advice, first I must acknowledge my own bias: Object-Oriented Programming (OOP) was the first design paradigm I was taught, and that will have a huge impact on how easy it is to get used to a Data-Oriented Design mindset. Many developers mention the difficulty of getting out of this OOP mindset and properly applying DOD, and this is a difficulty I experienced as well. As an attempt to stay somewhat unbiased, advice is mostly based on the information found in the manual, documentation, or forum threads/social media. However, much of the advice given is at least to some degree guided by my own experience while trying to write a prototype using the packages: Burst, Jobs, Entities and Physics. However, as this advice is written by a single person, personal bias will still influence the advice to some extent.

1. Recommendation to Rebels' Unity Team

DOTS packages, though most still in very early stages, are generally very powerful tools if used correctly in the right circumstances. However, because of missing features or documentation, instability, or lack of user friendliness I would generally not recommend it to be used as a tool for prototyping.

Use in production is recommended only if at least one of the programmers already has a lot of experience using DOTS, or if time is not of the essence. Some developers will grasp the core concepts of DOTS more quickly than others, and therefore time estimates become significantly more difficult.

Currently, the amount of programmers that use DOTS on a regular basis seems to be relatively low. New team members will most likely not have experience with DOTS yet; therefore maintaining old code could become significantly more difficult if the development team grows or changes. However, because ECS is built to have small, easily understandable systems that are separate from the rest of your code base, maintaining old code could actually be easier in some cases; it all depends on how well the developer implements DOTS/ECS.



Because Rebels' Unity team often has relatively small or short projects with high time pressure, **using DOTS would generally not be advised**. However, it would be beneficial to, whenever possible, invest time into learning how to implement the Job System and Burst, as multithreading is an important feature that the Job System makes relatively easy. Both of these packages are considered to be in the "Released" phase, which means that unlike other DOTS packages such as Entities, it is considered to be mostly feature complete and stable, will likely not change much in the future and will be compatible with newer versions, such as Unity 2021.

On the other hand, I would still recommend trying to at least implement the Entities package, as the ECS code structure seems to be preferable to most developers who use it, and this makes writing Jobs and Burst compatible code much easier with syntax improvements such as the *Entities.ForEach* structure. Because DOTS can be used together with GameObjects, this means there is effectively no downside to using DOTS at least to some extent, because any time that DOTS falls short you can simply implement whatever doesn't work using the old systems, until a DOTS implementation of your feature is available.

It is also a good idea to at least have some experience with DOTS, so that it can be used to solve performance issues if those were to occur. Keep in mind that though there is a steep learning curve, according to results to the survey writing Burst/Jobs code does not necessarily take more time than classic Unity/C# once a developer becomes used to these systems. Also, being aware of the way DOTS code is structured and keeping that in mind while creating your first prototype could save you a lot of refactoring down the line if you do ever decide you want to implement DOTS in your project.

Project Tiny is also an important part of DOTS to keep in mind in the future: while still at an early stage now, the prospect of being able to create tiny games is very interesting especially for companies like Rebels: the option to offer services such as playable ads or instant play games could be very valuable.

I would like to emphasize that despite all of the frustration I've expressed at certain parts of DOTS, I still believe DOTS is a great system, and **absolutely should not be ignored**, even in its current state. Despite DOTS still being in early stages and being a bit difficult to work with, the potential performance gains are incredible. DOTS is currently absolutely not perfect, and needs a lot before it can be used easily reliably by the majority of Unity developers, but it is very, very promising. Having a team that is at least capable of using these systems will give you a huge advantage over other Unity development teams.

Unity Technologies has also made it clear on multiple occasions they plan to one day completely replace GameObjects with Entities, though they do make it clear that they will keep supporting GameObjects as long as necessary.

If any version of Unity after 2020 LTS is needed for development, it is required to verify whether packages are currently expected to be compatible. Keep an eye on Unity's DOTS forum or other Unity communication channels for updates on compatibility and progress.

2. Recommendation to Rebels for a possible follow-up research

DOTS is a lot bigger than originally expected, and I came nowhere near exploring the full potential of DOTS, even though I did learn a lot. First of all, I don't believe follow-up research on the entirety of DOTS would be necessary in the near future. Once a new version of DOTS releases, I'd recommend going through the change logs and seeing if there are any significant improvements in usability, which there will most likely be a lot of. I recommend staying up-to-date on these developments, by for example going to Unity's DOTS forum to see if any developers have posted an update, or keeping an eye on Unity's social media or website.

However, waiting for a full release of a 1.0 version of DOTS before trying again might be best, as using current versions of DOTS can be very frustrating at times, and it might leave a bad impression if your team has to switch to DOTS before it is in a user-friendly state. I personally suspect this is the reason why Unity decided to go silent about DOTS for a while; I personally saw many posts and comments from developers who used DOTS in a very early stage, and are now possibly more reluctant to try a new version than they would be if they had never tried DOTS in the first place.

On the other hand, DOTS can offer such incredible improvements in aspects such as performance, control over code, et cetera, that I can't really recommend not using it at all either. Invest as much time as possible into learning DOTS during times when there is little to no work pressure, such as when you are between projects and have some extra time on your hands. ECS is a very nice structure, with a lot of advantages and apart from the learning curve, not many significant disadvantages.

Because Gijs Verheijen mentioned wanting research to be done on creating a build pipeline for Rebels' Unity projects, I would recommend doing a follow-up research on Burst's modding support capabilities, as this might allow for easier updating, or allow one part of the game to be included in the base download for a demo or first level, so you can play the game while it's being downloaded. It would be good to see if this is a better alternative to Unity's addressables system, or whether these systems provide some kind of benefit to each other. The Job System needs no more follow-up research in my opinion; there should be enough information in this document (or Unity's documentation) to properly get started using Jobs. The physics package will need some minor follow-up research if newer and more stable versions are released; as in my experience, handling collisions/triggers or modifying velocity are common use cases for a lot of video games and the current version of Unity Physics still seems very experimental and unstable.

3. Advice based on situation

This paragraph contains advice based on the findings and personal opinion of the author, but is more general advice to programmers who are interested in DOTS rather than specific advice to Rebels' current situation. Whatever your situation may be: at the very least keep an eye on DOTS updates, and keep DOTS in your mind as an option. DOTS will be the future of developing in Unity, and can cause incredible performance gains and flexibility.



To stay updated about DOTS, the best place is the [DOTS Forum](#), where the DOTS team shares updates about DOTS and sometimes reply to threads with some information about upcoming features. Subscribe to [Unity's Youtube channel](#) or other social media such as [Twitter](#), where they will most likely share promotional material, updates, or tutorials once production-ready versions of DOTS packages release. You can also subscribe to other Youtube channels that upload tutorials and updates about DOTS and Unity: "Turbo Makes Games" is a Youtube channel that mostly focuses on game development using DOTS, and is currently the largest source of up-to-date DOTS tutorials.

It is also good to already familiarize yourself with the concepts and restrictions of DOTS, and try to write code that at the very least can be more easily converted to DOTS, especially Jobs and Burst, by for example splitting up data stored in a MonoBehaviour's fields into multiple smaller structs that use blittable data types. Either way, you need to be aware that DOTS packages exist and in some cases are already usable; they can be a solution to a specific problem you have, such as bad performance or scalability.

If your priority is stability, user friendliness or quick development:

Do not use Entities for now. Keep an eye on updates around DOTS to see whether usability and user friendliness improves. Updates are currently infrequent; so there's no need to do this more than once a month. Jobs and Burst are stable, but might need some extra time to implement as there is a significant learning curve. However, most developers seem to report that it does not necessarily take extra time to use these packages once they get over the learning curve. The package Entities is also considered to be usable, but still needs some improvements and is not very user-friendly at the moment.

If you want a fully user-friendly experience: you might want to wait for Unity to release DOTS templates: when the new Universal Render Pipeline (URP) and High Definition Render Pipeline (HDRP) were ready for release, Unity released new templates in the Unity Hub, to start new projects that are immediately compatible with URP or HDRP. I would expect a similar template to work with DOTS systems. Getting DOTS to work in a project might currently seem like a tedious and unstable process, but when URP and HDRP were still in preview the same could be said for these packages. However, once the templates were released, these systems seemed to be mostly stable and were a lot easier to implement. It seems reasonable to assume that DOTS will follow a similar path.

If you're willing to spend some extra time on optimizing performance, but you require stability/production-readiness:

Implement Jobs and Burst wherever possible. If you are using a Unity version later than 2020 LTS, do not use any other packages as they are likely not currently supported. Entities is stable to some degree, but the use of other packages like Unity Physics, Unity Animation, and Netcode is not recommended. Using the Hybrid Renderer is also not recommended at the moment, though there are some developers who consider Hybrid Renderer to be usable for production.



If you can afford to build custom solutions for missing features, or deal with general instability:

Feel free to use any DOTS packages, especially Preview/Pre-Release or Verified/Released packages, as long as you're aware of the limitations and current development status. You might run into a lot of bugs, or won't have any documentation or community support to help when you're stuck. Experimental packages are also not necessarily guaranteed to be part of Unity's roadmap, meaning any experimental package you use might not receive any further updates or support in future Unity versions. Sometimes a DOTS package can completely change the capabilities of the Unity engine. DSPGraph, for example, gives you access to a low-level audio renderer. This makes it a great package to explore for electronic musicians or sound engineers. When using any experimental package, make sure to spend a lot of time on whatever subforum is most relevant to that package; simply reading threads posted by other DOTS users can be surprisingly educational.



1. What is DOTS?

This chapter will explain the packages that are the “core” of DOTS: **Entities**, **Jobs**, and **Burst**. It will also discuss a package I personally found important for prototyping: **Unity Physics**.

Unity's GitHub page contains [a repository with DOTS sample projects](#). The “readme” file of this repository explains the three core packages of DOTS:

“We have been working on a new high performance multithreaded system, that will make it possible for games to fully utilise the multicore processors available today without heavy programming headache. The Data Oriented Tech Stack includes the following major systems:

*The **Entity Component System** provides a way to write performant code by default.*

*The **C# Job System** provides a way to run your game code in parallel on multiple CPU cores*

*The **Burst compiler** a new math-aware, backend compiler tuned to produce highly optimized machine code.*

With these systems, Unity can produce highly optimised code for the particular capabilities of the platform you're compiling for.”

They call this “Performance by Default”. In short, DOTS is a collection of packages that mostly aim to increase performance and/or flexibility.

The following paragraphs will explain some, but not all, of the core concepts of DOTS. DOTS is very large and complex, and therefore explaining DOTS completely wouldn't be possible in the amount of time spent researching, or even useful as this document is already fairly long. Furthermore, an attempt at fully documenting DOTS would essentially amount to a differently worded version of the already extensive Manual pages. For these reasons, this section instead is a selection based on my own experience and research and is in no way a full explanation; for a more complete description of the packages please refer to [Unity's page about DOTS packages](#) to find the manual pages and documentation.

1.1. Burst and the DOTS Runtime

The Burst Compiler is a new compiler for Unity that can compile Unity code into very high performant machine code, but has some caveats to its use. It is made to work with Jobs and ECS, so implementing those packages correctly should result in Burst-compatible code.



Usually, compiling code for Unity does not generate native code directly, rather it first goes through an intermediate language. Mono uses the Common Intermediate Language (CIL), while IL2CPP compiles to C++. The Burst Compiler can, instead, compile C# directly to native code. This gives Unity developers more control over compilation. However, it relies on a subset of C# that has many features stripped that would otherwise impact performance. The blog [On DOTS: C++ & C#](#) from 2019 explains the decision to move towards this new subset of C#, which they refer to as **"High Performance C#" (HPC#)**:

The C# language team, standard library team, and runtime team have been making great progress in the last two years. Still, when using C# language, you have no control over where/how your data is laid out in memory. And that is exactly what we need to improve performance.

On top of that, the standard library is oriented around "objects on the heap", and "objects having pointer references to other objects".

That said, when working on a piece of performance critical code, we can give up on most of the standard library, (bye Linq, StringFormatter, List, Dictionary), disallow allocations (=no classes, only structs), reflection, the garbage collector and virtual calls, and add a few new containers that you are allowed to use (NativeArray and friends). Then, the remaining pieces of the C# language are looking really good.

In more recent versions of Burst (currently incompatible with the latest the DOTS packages), there are some interesting new features, such as intrinsics and modding support.

Intrinsics

The Burst manual has some pages about how you can optimize this code, such as the page "[Optimization Guidelines](#)". One of the ways this can be achieved is through "intrinsics": these optimize branching by, for example, telling the compiler that one branch is more likely than the other.

```
if (Unity.Burst.CompilerServices.Hint.Likely(b))
{
    // Any code in here will be optimized by Burst with the assumption that we'll probably get
    here!
}
else
{
    // Whereas the code in here will be kept out of the way of the optimizer.
}
```




Modding support

Burst now has [modding support](#), which allows you to release a part of your project so that others can add functionality to your game themselves using the Unity editor. Users can compile this code themselves and add it to a folder in the games' install directory. Developers can define the parts of their game that can be modified by releasing interfaces or base classes to the public and internally implementing a "mod manager" to define the way these interfaces change your program.

DOTS Runtime

The DOTS Runtime allows Unity to compile pure DOTS applications to be extremely small, by excluding unused parts of the Unity engine. Compiling an application to be under 15MB can, for example, allow you to publish your game to [Google Play Instant](#). The term "DOTS Runtime" seems to be used somewhat interchangeably with [Project Tiny](#), the first experimental package that used this runtime.

Job System

The Unity Job System is one of the oldest, and most important, parts of DOTS. The Job System makes it possible to define code as a "Job". Jobs are small pieces of code that make **multithreading** in Unity easier.

The problem that the Unity Job System solves is the occurrence of "race conditions": a very common problem with multithreading. Race conditions are unpredictable circumstances that often lead to bugs, which are caused because multithreading makes the execution order of code unpredictable. The debugging of these race conditions can sometimes be very difficult because the placement of breakpoints can affect the timing of threads that cause these race conditions.

To address this issue, the Job System only works on copies of data, instead of the data itself. Because of the way this system copies data, jobs only work with **blittable data** types; this ensures that blocks of data can be copied using the memcpy function. From [The safety system in the C# Job System](#):

To make it easier to write multithreaded code, the Unity C# Job System detects all potential race conditions and protects you from the bugs they can cause. For example: if the C# Job System sends a reference to data from your code in the main thread to a job, it cannot verify whether the main thread is reading the data at the same time the job is writing to it. This scenario creates a race condition. The C# Job System solves this by sending each job a copy of the data it needs to operate on, rather than a reference to the data in the main thread. This copy isolates the data, which eliminates the race condition. The way the C# Job System copies data means that a job can only access blittable data types. These types do not need conversion when passed between managed and native code.

Entities:

The Entity-Component-System pattern:

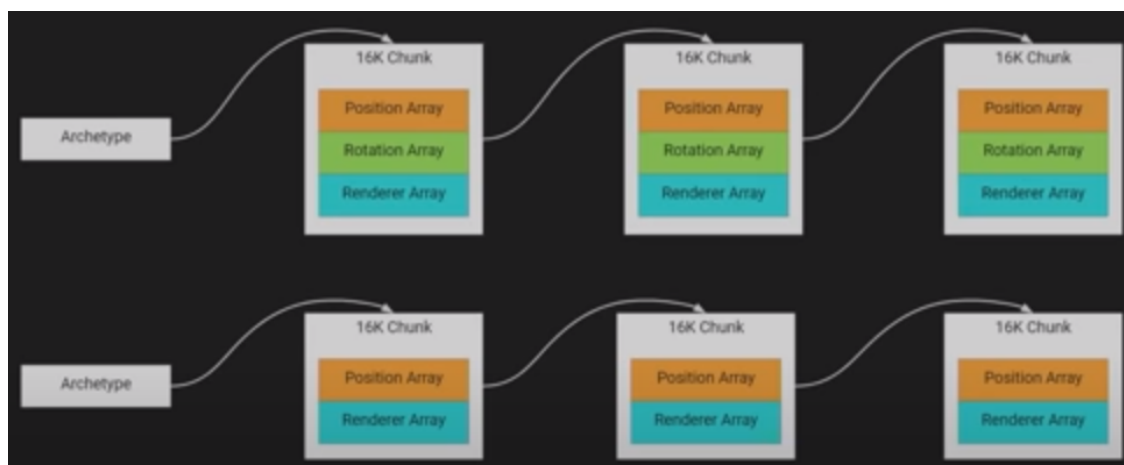
The **Entity-Component-System (ECS)** design pattern has been popular with game development for a while. ECS separates your program into 3 distinct categories:

- **Entities** are somewhat equivalent to GameObjects. However, they do not actually contain any data or behaviour.
- **Components** are your data; they are attached to an entity.
- **Systems** define behaviour. They generally iterate over entities by filtering entities based on the types of components that are attached to those entities.

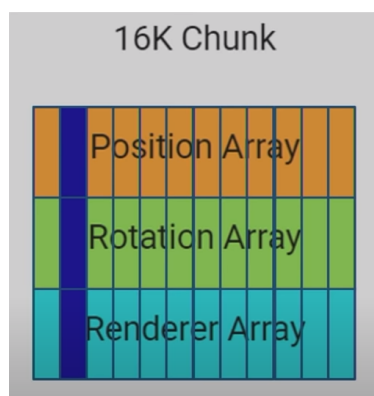
Data layout: Chunks, Archetypes and cache misses

An **Archetype** is a set of component types. It is kind of like a "blueprint" for an entity, not containing any data but defining the structure of one type of entity. For example, one archetype "A" could be defined as having a Translation(position) and a Rotation component, and an archetype "B" could be defined as having Translation, Rotation, and Scale. If an entity belongs to archetype A, and a Scale component gets added to this entity, the entity will automatically move to archetype B. If another component is added, it would move to a new archetype, "C".

A good explanation about how archetypes and chunks work can be found in the presentation ["C# Job System + ECS usage and demo with Intel - Unite LA"](#) (32:34-42:24). Essentially, entities are grouped together in memory based on their archetype. They are stored in **chunks**. Any change that would change the archetype of an entity, such as creating/destroying an entity or adding/removing a component, therefore changes your memory layout. This is called a **structural change**.



A chunk is a collection of arrays that store a set of objects of the same archetype. For example, the image below shows what a chunk would look like for an object with the components "Position", "Rotation", and "Renderer". There are 12 total objects stored in this chunk. The objects are defined by their index in these arrays: for example, the second object in this chunk would have the component at the second index. The size of the arrays within a chunk is therefore always equal to other arrays within that chunk, in this case a size of 12..



A good way to think about DOTS is as if it is a database: you query the database for a set of tables, perform some kind of calculation on its data, and write the results back into your database.. From ["Understanding data-oriented design for entity component systems - Unity at GDC 2019"](#):

If it looks an awful lot like a database, that's because it is. Having an in-memory database lets us query for specific sets of component data and we call these component type sets just "archetypes". And they actually kind of serve as a functional equivalent to objects [...] but without the mental and computational overhead.

Mike Geig also makes this comparison to a database when talking about the difficulty in getting used to a DOD mindset, compared to OOP:

The most difficult thing is just sort of wrapping your head around the different way of thinking about things. It's not intuitive because humans don't think about stuff like that. we look at something and go 'That's a car'. We don't look at that and go "That's the color red. Number of wheels: four. Doors: four." [...] Even in schools we all learn Object Oriented Programming, we don't really learn Data Oriented Programming. So the best way to think about this is to think about DOTS as if it's a database and you're just running queries against the database. Find me this data, do something with the data, put it back. Think about DOTS as if it was a shader for your CPU, right? The reasons you would do things on CPU versus a shader on your GPU: same reasons of MonoBehaviours versus DOTS. These are kind of the same concepts where we're streamlining and thinking about data just like we would with a shader.

Physics: Stateless physics

The new package "Unity Physics", besides being the Entities-compatible replacement of the old physics system, is a **stateless** physics system. Unity's manual page has a section, ["Design philosophy"](#), explaining this decision:

Modern physics engines maintain large amounts of cached state in order to achieve high performance and simulation robustness. This comes at the cost of added complexity in the simulation pipeline which can be a barrier to modifying code. It also complicates use cases like networking where you may want to roll back and forward physics state. Unity Physics forgoes this caching in favor of simplicity and control.

They also mention the modular nature of this physics engine, as developers are able to manually manage physics time steps, and even execute multiple physics steps simultaneously to, for example, predict the path an object will take or predict future collisions.

In Unity Physics, core algorithms are deliberately decoupled from jobs and ECS. This is done to encourage their reuse as well as to free you from the underlying framework and stepping strategy. As an example of this, see the Immediate Mode sample. It steps a physical simulation multiple times in a single frame, independent of the job system or ECS, to show future predictions.

Features

It seems that at some point, all important Unity features should have some sort of DOTS equivalent. Unity has repeatedly said that **the plan is to one day completely convert the workflow from GameObjects and MonoBehaviours to Entities**. The DOTS Runtime used by Project Tiny is a complete replacement of the classic *UnityEngine* namespace: according to a blog post by Unity, at some point even the entire Unity engine will be rewritten in High Performance C# (HPC#); the developers apparently find HPC# easier to debug and manage than C++. In some cases, they even saw an increase in performance after porting certain systems from C++ to HPC#.

From the 2019 blog post by Lucas Meijer "[On DOTS: C++ & C#](#)":

We're sometimes faster than C++, also still sometimes slower than C++. The latter case we consider performance bugs we're confident we can resolve.

Only comparing performance is not enough though. What matters equally is what you had to do to get that performance. Example: we took the C++ culling code of our current C++ renderer and ported it to Burst. The performance was the same, but the C++ version had to do incredible gymnastics to convince our C++ compilers to actually vectorize. The Burst version was about 4x smaller.

[...]

For most of us, it feels like "you're closer to the metal" when you use C++. But that won't be true for much longer. When we use C# we have complete control over the entire process from source compilation down to machine code generation, and if there's something we don't like, we just go in and fix it.

We will slowly but surely port every piece of performance critical code that we have in C++ to HPC#. It's easier to get the performance we want, harder to write bugs, and easier to work with.

Pros & cons

This section will discuss the pros and cons of using DOTS. However, always keep in mind that DOTS is still very new, and many things will most likely improve in the future. As this advice is written by a single person, some degree of personal bias in this list is to be expected. However, my opinions do seem to mostly align with DOTS forum users, according to the results of the [DOTS survey](#). This survey and the pros/cons described by Unity Technologies and other ECS users such as [Tim Ford from Blizzard Entertainment](#) have been the prominent sources used to create this list.

Generally, DOTS packages are made to improve **performance** and/or **flexibility**. Though most of them need much work to be done when it comes to missing features and user friendliness, the ones that are already usable are very impressive. I, like many other DOTS forum users, am convinced that one day DOTS will be the default method of game development in Unity. However, the progress seems to be slow at the moment; predicting how quickly this will happen is impossible. It looks like Unity version 2021 will not be supporting DOTS; we might have to wait for next year or even later for a 1.0 release, or even a next update.

Some of the pros and cons per package are:

Jobs

Pros:

- Better performance through enabling a safe **multithreading** system
- According to the survey, writing Jobs does not take longer than normal C#/MonoBehaviour code once you are used to

Cons:

- Can only use **blittable** types, disabling features like classes.

Burst

- Huge increase in performance, seemingly the biggest contribution to the high performance in DOTS packages
- More control over compilation, such as optimizing branches
- Modding support
- No more Garbage Collector, because the GC only works on managed types. This results in a lack of "GC spikes", a somewhat common performance issue.

Cons:

- Only works on **unmanaged** types, disabling many features and types like classes, strings and arrays.

Entities

Pros

- Better performance through **Data Oriented Design (DOD)**
- The **Entity-Component-System (ECS)** structure allows for code that is more easily reusable, maintainable and readable.
- Has the least restrictions of the three core DOTS packages; you can use managed types and in theory, you can use ECS with GameObjects. However, it works even better with Jobs and Burst, and makes it easier to write code that is compatible with these packages.
- ECS prevents complex inheritance hierarchies. The reusability of ECS code makes up for the lack of inheritance, polymorphism, et cetera.

Cons

- To developers who are used to Object-Oriented Programming, DOD can be challenging at first, making time estimates difficult.
- Though it is not necessarily more difficult, it is different. Because not many people use ECS/DOTS currently, finding resources is much more difficult than for classic Unity.

2. How to implement DOTS

This chapter will serve as an instruction manual on how to use DOTS, find sources, and avoid issues I personally ran into when exploring DOTS systems. I am not affiliated with Unity Technologies in any way; therefore some may prefer to get the information straight from the source. Unity Technologies already has extensive documentation for most important DOTS packages (excluding most packages that are marked as "Experimental"), and has also released several tutorials on their YouTube channel and Unity Learn platform. **Refer to the chapter [Important sources for an overview of links to Unity's own tutorials, manuals and documentation](#), and other helpful third party information sources.**

Because DOTS is a very large and complicated subject and time for research was limited, only the most important packages were included in the research:

- Entities
- Jobs
- Burst
- Physics

How to implement Physics is currently not included however, because I still expect some major changes to its syntax. For examples on the current way you should implement Physics, you can see some in the attached [prototype](#), or Unity's [ECS samples](#).

Besides these packages, some others are mentioned briefly. However, these packages either were considered low priority, or were excluded from the research entirely. The amount of information about the aforementioned packages is still too large to list in this document. Instead, a [list of resources](#) is included at the end of this document. These are a collection of different resources for information about DOTS, such as official tutorials and courses or manual pages. However, many DOTS packages, such as Animation and Project Tiny, still lack this information.

Some basic information about implementation will be shared, though this is mostly a summary of information that was most useful while trying to write a simple prototype using these packages and is in no way an extensive report of DOTS functionality. **For a much longer and more complete overview you should always refer to Unity's official manual pages and documentation.**

While writing this document, I wrote some DOTS code as an attempt to prototype a small game as a research method. Though it gave me a lot of insight on DOTS systems, this is a very basic prototype. It contains some code samples that may be useful, such as personal solutions for hybrid implementations, or scenes demonstrating certain functionalities, restrictions or bugs/errors a developer should be aware of. This prototype can be found on GitHub: https://github.com/Rebels-io/DOTS_Prototype



I would like to reiterate this is in no way a complete guide to DOTS. This is an advisory report written for developers in the Unity Team at [Rebels](#). It will show them how to get started with developing DOTS, and go over some of the topics that were useful in my own attempts at prototyping using DOTS, but more importantly it contains context-dependent advice specifically for Rebels.

Installing the correct Unity editor version and packages

Finding the right combination of packages without causing incompatibility issues can be challenging when it comes to DOTS. This paragraph will contain instructions on how to install the Unity Editor and DOTS packages. The Unity version/packages used for the prototype are:

Unity Editor: 2020 LTS (the specific version used is **2020.3.12f1**, but any version of Unity 2020.3 should work)

Entities (com.unity.entities): **0.17.0-preview.42**

Burst (com.unity.burst): **1.4.1**

Jobs (com.unity.jobs): **0.8.0-preview.23**

Hybrid Renderer (com.unity.rendering.hybrid): version **0.11.0-preview.44**

Physics (com.unity.physics): **0.6.0-preview.3**

DOTS Editor (com.unity.dots.editor): version **0.12.0-preview.6**

Platforms Windows (com.unity.platforms.windows): **0.10.0-preview.10**

Step one: Install the Unity Editor

First, install the Unity Editor. Unity's DOTS team currently recommends you to use **Unity version 2020 LTS** (Long Term Support) if you want to use DOTS. Though certain packages (those marked as "Released") are currently compatible with Unity version 2021.1, such as Burst and Jobs, [others are expected to be incompatible](#):

You must stay on Unity 2020 LTS and on Entities 0.17 for now. Future releases of Entities will not be compatible with Unity 2021 until the end of the year at the earliest. Upgrading to 2021.1 and using current or future Entities packages will not work and is not expected to work.

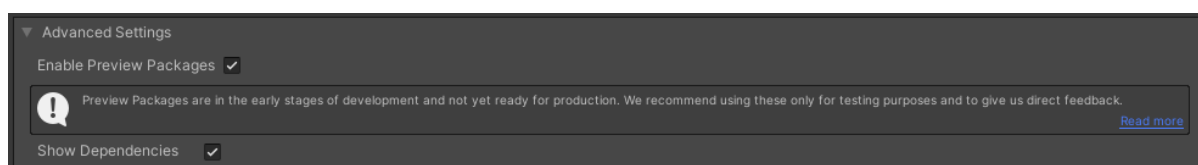
To install this version, first install the Unity Hub. Then, using the Unity Hub you can install the latest 2020.3 release.

Step two: Set up your project and scene

Create a new project. Unless Unity has already released a DOTS template project by the time you read this, create a project from the Universal Render Pipeline (URP) or High Definition Render Pipeline (HDRP) template. Using the default render pipeline (by using the 2D or 3D template) seems to still be supported for now, however in this document it is assumed that you will be using URP or HDRP. If this is not the case, there might be some differences in the way the Hybrid Renderer works.

Step three: Install DOTS packages

Because many DOTS packages are considered Preview (or Pre-Release) packages, these should be enabled in your settings first. In Unity version 2020 LTS, this setting can be found in *Project Settings > Package Manager > Advanced Settings > Enable Preview Packages*.



However, since Unity version 2020.1 the Hybrid Renderer package, and other DOTS packages, are a bit harder to find: [some preview/experimental packages are no longer found in the Package Manager even when enabling these options](#) and need to be added manually. As mentioned previously, you also need to make sure that your Unity version is compatible with these packages. As mentioned in the previous chapter, DOTS consists of three “core” packages: **ECS, Burst, and Jobs**. Installing the latest version of the **Hybrid Renderer** package in the Package Manager, because of the dependencies of this package, will automatically install the correct versions of these packages.

Install the following packages:

Hybrid Renderer:	com.unity.rendering.hybrid
Unity Physics:	com.unity.physics
DOTS Editor:	com.unity.dots.editor

After installing these packages, all of the DOTS packages should be installed and there shouldn't be any version incompatibility. It is also recommended to enable [Hybrid Renderer V2](#), which has better performance and more features than the default Hybrid Renderer.

Step four: Settings for increased editor performance

Debug settings

When DOTS packages are installed, there are a few options to allow for better debugging of DOTS systems. If your performance isn't as good as you would expect, odds are you might have forgotten to disable one of these features. These features are described in [Best Practices chapter 3.1: Implementation fundamentals](#):

Jobs > Leak Detection > On: This warns you about potential memory leaks that relate to the allocation of Native Containers. Alternatively, enable **Jobs > Leak Detection > Full Stack Traces (Expensive):** to track down the causes of specific leaks. Leak Detection works by making and tracking small managed allocations, so if you see lots of GC Allocs in the profiler, disable this option and try again.

Jobs > Burst > Safety Checks: This warns you about out-of-bounds errors with Native Containers and issues with data dependencies

Jobs > JobsDebugger: This warns you about issues that relate to job scheduling and conflicts

Unity Editor Status Bar > Debug Mode: This setting allows you to attach a debugger to the Editor while it's running. From Unity 2020.1 onwards this option is in the icons at the bottom right of the Unity Editor window. In earlier versions of Unity, the option is in Preferences > External Tools > Editor Attaching

Because these features significantly reduce performance, you should always disable these unless absolutely necessary. Also, make sure that you actually enable Burst compilation in your project by finding the setting in *Jobs > Burst > Enable Compilation*.

Play mode settings

Joachim Ante also posted a [thread on the Unity DOTS forum](#) regarding settings which allow you to enter Play Mode faster. These are the recommended settings:



According to the [blog post](#) Ante is referring to in this thread, this can save up to 50-90% of waiting time.

Implementing the Jobs package

Using the Entities package allows you to use the `Entities.ForEach` structure to automatically create jobs. However, a Job can also be defined without using the Entities package by implementing the `IJob` interface.



```
// Create a native array of a single float to store the result. This example waits for the job to complete for illustration purposes
NativeArray<float> result = new NativeArray<float>(1, Allocator.TempJob);

// Set up the job data
MyJob jobData = new MyJob();
jobData.a = 10;
jobData.b = 10;
jobData.result = result;

// Schedule the job
JobHandle handle = jobData.Schedule();

// Wait for the job to complete
handle.Complete();

// All copies of the NativeArray point to the same memory, you can access the result in "your" copy of the NativeArray
float aPlusB = result[0];

// Free the memory allocated by the result array
result.Dispose();
```

```
// Job adding two floating point values together
public struct MyJob : IJob
{
    public float a;
    public float b;
    public NativeArray<float> result;

    public void Execute()
    {
        result[0] = a + b;
    }
}
```

Because Jobs copy and isolate data, the results are isolated as well. Storing the results of a Job should be done using a [NativeContainer](#).

Implementing the Burst package

Implementing Burst is very similar to Jobs; you can't use managed types. As blittable types are already not managed, Burst should always be able to compile any Jobs that are meant to run on worker threads. Adding the [BurstCompile] tag should ensure that your function is compiled by Burst; however when using other DOTS packages such as Entities, in many cases this is automatic behaviour.

Also, you do need to actually enable Burst in your project after installing the package, by using the setting "*Jobs>Burst>Enable Compilation*" in the Unity editor. Read the [Burst Manual](#) for more information.

Math

Whenever you need to use a math function in a burst-enabled piece of code, you should use the new *Unity.Mathematics* namespace instead of the old *UnityEngine.Mathf*. The syntax is slightly different, as it is supposed to be more familiar to those with experience in programming with SIMD or Shaders/graphics. For example, a *Vector3* value is stored as a *float3* instead.



Collections

DOTS uses NativeArrays, which is a wrapper pointing to unmanaged memory. NativeArrays can be used to share data between managed and unmanaged code, such as storing the results of a Job to use on the main thread.

Implementing the Entities package

Component data types

Data in ECS is all stored components; there are different types of components.

ComponentData

ComponentData, often called a "runtime component" or simply "component", is the ECS equivalent of a GameObject's component. The major difference between a ComponentData and a MonoBehaviour is that ComponentData only contains data, whereas a MonoBehaviour usually contains both data and code. You can use *[GenerateAuthoringComponent]* to use your component data with the entity conversion workflow; refer to the paragraph "[Authoring components](#)".

To define a ComponentData, you define a **struct** that inherits from the interface **IComponentData**. To attach it to an entity, you can use an *EntityManager* and call the methods *AddComponentData* and/or *SetComponentData*.

You can also make a ComponentData a **class** instead of a struct. This is called a "**managed component**". This makes it incompatible with multithreading and Burst compilation, requiring you to disable Burst with *[BurstDiscard]* or *WithoutBurst* and run any code using this managed component on the main thread by using *.Run* instead of *.Schedule/.ScheduleParallel*. This makes it easier to implement a hybrid solution, for example if you need to use a feature that has no DOTS equivalent yet, such as animation or UI: this [video tutorial by Johnny Thompson](#) shows how to use a managed component to implement UI.



DynamicBuffer

A [DynamicBuffer](#) is very similar to a `ComponentData`, but works like a list. It provides methods such as `.Add` and `.Remove`. To define a `DynamicBuffer`, you first define the individual elements of this buffer as a **struct** that inherits from the interface

IBufferElementData.

```
public struct MyBufferElement : IBufferElementData
{
    // Actual value each buffer element will store.
    public int Value;

    // The following implicit conversions are optional, but can be convenient.
    public static implicit operator int(MyBufferElement e)
    {
        return e.Value;
    }

    public static implicit operator MyBufferElement(int e)
    {
        return new MyBufferElement { Value = e };
    }
}
```

To attach it to an entity, you use an `EntityManager`. In the example shown above, the correct syntax would be `EntityManager.AddBuffer<MyBufferElement>(entity)`.

You can access it in your entity query by using the type `DynamicBuffer<T>`, where `T` is your `IBufferElementData`.

```
public partial class DynamicBufferSystem : SystemBase
{
    protected override void OnUpdate()
    {
        var sum = 0;

        Entities.ForEach((DynamicBuffer<MyBufferElement> buffer) =>
        {
            for(int i = 0; i < buffer.Length; i++)
            {
                sum += buffer[i].Value;
            }
        }).Run();

        Debug.Log("Sum of all buffers: " + sum);
    }
}
```



SharedComponentData

A SharedComponentData is a specific type of component that actually changes the archetype of your entity. Entities that have the same set of components, but don't have the same value on every SharedComponentData, will be considered a separate archetype. You can filter sharedComponentData in the same way you'd filter a component; by defining it as part of the archetype (for example, by using `EntityQuery.SetSharedComponentFilter()`)

Systems

To create a System, you simply create a class that inherits from `SystemBase`. You will need to implement the `OnUpdate` function. This function runs every frame as long as your system is enabled and if it contains an `EntityQuery` that is not currently empty (or if you've added the tag `!AlwaysUpdateSystem!`).

Entity queries

Systems generally iterate over a set of entity components to process their data in some way. From [Unity's manual page on entity queries](#):

You can use an EntityQuery to do the following:

- Run a job to process the entities and components selected
- Get a `NativeArray` that contains all of the selected entities
- Get `NativeArrays` of the selected components (by component type)

This page shows several examples of how to define an entity query, such as by using the class `EntityQueryDesc`:

```
var queryDescription = new EntityQueryDesc
{
    None = new ComponentType[] { typeof(Static) },
    All = new ComponentType[] { typeof(RotationQuaternion),
                                ComponentType.ReadOnly<RotationSpeed>() }
};
EntityQuery query = GetEntityQuery(queryDescription);
```

However, you'll generally generate these queries automatically when you use the `Entities.ForEach` syntax.

.WithoutBurst, .Run, and .WithStructuralChanges

If you ever run into a situation where you need to debug your code or you need to use something that is not compatible with Burst (for example, if you're working with a non-blittable data type), you can choose to disable Burst from a single system by using the `.WithoutBurst()` method.

```
Entities
    .WithoutBurst()
    .ForEach((Entity e, GUIDReferenceSenderComponentData s) => {
```

Note that you do not need to run code on the main thread using *.Run* to enable debugging with *UnityEngine.Debug.Log*; you only need to disable Burst to do this.

```
Entities
    .WithoutBurst()
    .ForEach((ref TestComponent t) =>
    {
        t.Val++;
        Debug.Log("Running GameObjectForEachTestSystem: ForEach(Entity)");
    })
    .Schedule();
```

If you need to make structural changes, such as adding or removing components/entities, you would generally want to use an *EntityCommandBuffer* as the *EntityManager* cannot be accessed from a *Job*, but you can also simply both disable Burst and run code on the main thread. To make this a bit easier, there is the *.WithStructuralChanges* method, which is the same as using *.WithoutBurst* and *.Run*.

Entities.ForEach and Job.WithCode

The class **SystemBase** gives you the option to query entities using **lambda** functions using the **Entities.ForEach** structure. This is the recommended way to develop ECS systems according to the [manual page](#). You simply define the set of components you want, and then any entity that has that set of components will be included in the query.

```
Entities
    .ForEach((ref Translation translation,
              in Velocity velocity) =>
    {
        translation.Value += velocity.Value;
    })
    .Schedule();
```

You have to define whether your components are read-only or read+write, by using the **in** and **ref** keywords respectively in your lambda function. Your IDE might not autofill struct properties within the lambda function; a possible solution is using a static function call instead of defining your code as a lambda; don't forget to include *in/ref* keywords in these functions either.

Filtering the Entities.ForEach query

You can further filter your query with LINQ-style clauses such as *WithAll* or *WithNone*. The page for [SystemBase in the API documentation](#) contains a list of clauses to add to your query.



You can use `Jobs.WithCode` to run code if the data you are working with is not currently associated with an Entity; the syntax is similar to the `Entities.ForEach` query, but does not define a set of components. The manual has a page about [Jobs.WithCode](#), which shows an example of this syntax:

```
Random randomGen = new Random(seed++);
NativeArray<float> randomNumbers
    = new NativeArray<float>(500, Allocator.TempJob);

Job.WithCode(() =>
{
    for (int i = 0; i < randomNumbers.Length; i++)
    {
        randomNumbers[i] = randomGen.NextFloat();
    }
}).Schedule();
```

Using MonoBehaviours with `Entities.ForEach`

Disabling Burst and running `Entities.ForEach` on the main thread will also allow you to use managed components, including `MonoBehaviours`, as part of the `ForEach` EntityQuery. This doesn't enable the same performance gains as iterating on entity components would, but this does make synchronizing `GameObject` and Entity data easier. Sometimes this is the best way to implement a hybrid solution. This can also be used to keep using `MonoBehaviours` while also maintaining the separation of data and code/systems that the ECS provides.

To do this, your component needs to be associated with an Entity. One way to do this is to use the `ConvertToEntity` component with the "Convert and Inject" option selected. Note that this will likely change, as currently the `ConvertToEntity` component is planned to be deprecated. You can also manually add Object classes (including `MonoBehaviour`) to an entity using `EntityManager.AddComponentObject(Entity, Object)`. There are no plans for deprecating hybrid solutions, including methods such as `AddComponentObject`.

```
Entities
    .WithoutBurst()
    .ForEach((GameObjectTestBehaviour b) =>
    {
        b.Val++;
        Debug.Log("Running GameObjectForEachTestSystem: ForEach(Main Thread)");
    }).Run();
```




Entity interaction (data relationships):

One of the presentations given during the event Unite Copenhagen in 2019 explains how to think about the ways entities interact with each other (or more accurately, the way to think about data relationships) in ECS: "[Options for Entity interaction - Unite Copenhagen](#)"

- **Read relationship**
 - The component data of the entity we iterate on depends on the component data of another entity
- **Write relationship**
 - The component data of the entity we iterate on is used to write data to another entity's component

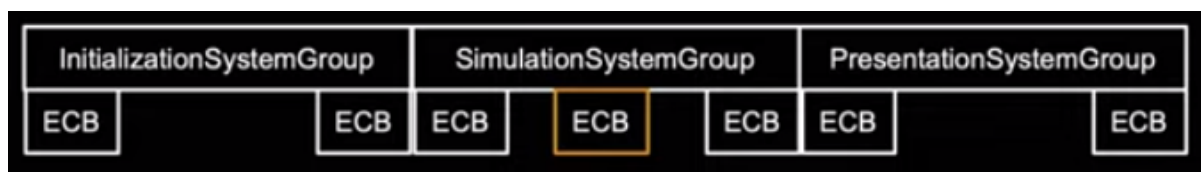
The best way to implement your relationship is mostly dependent on the **frequency**, and can be different for every use case: use the profiler to determine what would be the ideal implementation.

ComponentDataFromEntity

- Usually the right tool for implementing **read relationships**
- It's the tool to access component data of an arbitrary entity
- **Can be random memory access!** This means a potentially expensive read operation (cache miss)

EntityCommandBuffer (ECB)

- Usually the right tool for implementing **write relationships**.
- You can plan for when a certain change, like adding a component, takes place. There are default ECB systems at the start and end of every SystemGroup, e.g. "SimulationSystemGroup" has "BeginSimulationEntityCommandBufferSystem" and "EndSimulationEntityCommandBufferSystem". You can also define custom systems and the exact time of execution (with *[UpdateBefore]*, *[UpdateAfter]*, etc). However, this is not usually needed or recommended, as creating a custom CommandBufferSystem will introduce a sync point.



For example, the following line of code will create a reference to the third default ECB, BeginSimulationEntityCommandBufferSystem

```
BeginSimulationEntityCommandBufferSystem beginSimECB = World.DefaultGameObjectInjectionWorld
    .GetOrCreateSystem<BeginSimulationEntityCommandBufferSystem>();
```



GameObject to Entity Conversion

The [Conversion Workflow](#) is the current way Unity Technologies is trying to bridge the gap between implementing MonoBehaviours/GameObjects and ECS/DOTS.

GameObjects are still created in the same way, but the GameObject converts to an Entity and the GameObject either gets destroyed or becomes a hybrid GameObject/Entity, depending on the selected conversion mode (Convert and Destroy vs Convert and Inject).

Authoring components

To convert a custom MonoBehaviour to a DOTS component, you can implement the interface ***ICConvertGameObjectToEntity***. You will need to implement the function `void Convert(Entity, EntityManager, GameObjectConversionSystem)`. This is referred to as an **authoring** component: meaning that it exists only to define the contents of a DataComponent, and does not exist at runtime.

```
public class LifetimeAuthoring : MonoBehaviour, ICConvertGameObjectToEntity
{
    [SerializeField] float TotalLifeTime;
    10 references
    public void Convert(Entity entity, EntityManager dstManager,
        GameObjectConversionSystem conversionSystem)
    {
        dstManager.AddComponentData(entity, new Lifetime(TotalLifeTime));
    }
}
```

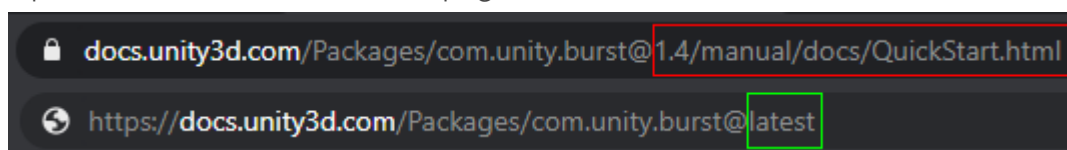
The example shown above is the simplest authoring component you can define; the runtime data (the type *LifeTime*) contains only a single field, and the authoring contains a single field with the same type, storing it without modifications. You can add the `[GenerateAuthoring]` attribute to a runtime component struct to automatically generate this authoring component. However, it can also be convenient to create a single custom authoring component to define multiple runtime components instead of generating them automatically.

Also, note that when you're implementing the *ICConvertGameObjectToEntity* interface, you don't know the order in which components are converted. This means you can't always read the value of a component that should be attached to the Entity in the overridden `Convert` method, because it might not be attached yet. If that is necessary, you can create custom systems by inheriting from the class `GameObjectConversionSystem`, and define its conversion order by using tags such as `[UpdateBefore]`/`[UpdateAfter]`.

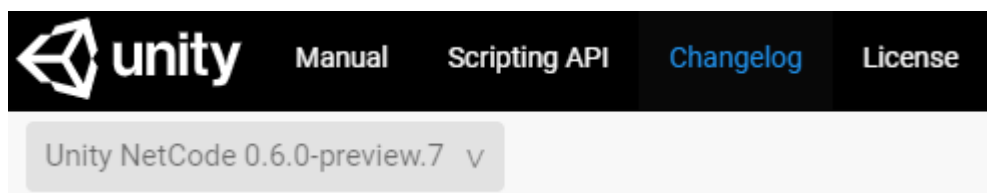
Finding up-to-date DOTS information

Using a search engine (e.g. Google, DuckDuckGo) is usually an easy way to quickly find documentation or forum threads when you're looking for more information or solutions. However, outdated syntax may not be valid. For example, you might come across a tutorial that uses an `Entities.ForEach` function without `.Run`, `.Schedule` or `.ScheduleParallel`. This is outdated syntax; you will get a runtime error telling you you need a `.Run`, `.Schedule` or `.ScheduleParallel` added to the end of the `ForEach` statement.

Another important thing to remember is that search engines might not always show the last versions of documentation. One trick that can help you find the correct version is to replace the version number and page in the url with "latest".



This will bring you to the homepage of the latest version of the package; use the search bar on this page to find the page you're looking for in the documentation of this version. If the page you're looking for cannot be found in the latest documentation, there is a chance that whatever you're looking for might be deprecated. In this case, you can try to find this in the changelog to see if it has been renamed or replaced. You can find a link to the changelog in the top bar of the documentation page of a package.



Building DOTS projects

To build a DOTS project, you need a separate package depending on your target platform. From [Unity's Entities manual](#):

Making standalone builds of DOTS projects requires installing the corresponding platform package for each of your target platforms:

*com.unity.platforms.android
com.unity.platforms.ios
com.unity.platforms.linux
com.unity.platforms.macos
com.unity.platforms.web
com.unity.platforms.windows*

After installing the platform packages you need, create a "Classic Build Configuration" asset for each platform (via the "Assets > Create > Build" menu). The properties of that asset will contain a "Scene List", which is the only way of adding subscenes to a standalone project. Make sure you add at least one scene or that the "Build Current Scene" checkbox is toggled on.

WARNING

Do not use the Build and Run menu "File > Build and Run" to build DOTS projects. It might work in some cases but this approach is not supported. You must build your project using the Build or Build and Run buttons at the top of the Build Configuration Asset Inspector window.

Installing (one of) these packages will most likely require you to [install them manually](#).

In this Build Configuration menu, you can add a LiveLink component to make the build compatible with LiveLink, allowing you to update builds remotely without having to restart the application on your target hardware.

3. Personal experience and opinions

This chapter will document the experience I had trying to learn and implement DOTS systems. Therefore, it will contain a lot of personal opinions rather than facts, and it should be taken as such. These opinions do not represent Rebels or any other person or entity other than the author unless explicitly stated otherwise, for instance if a source has been referenced or quoted.

Personal opinion about DOTS

DOTS is, in my opinion, probably the most interesting improvement Unity has had in years. Personally, I've always heard complaints about Unity's performance compared to other engines like C# as one of the primary reasons not to use Unity: it seems to have a reputation as an engine for generally lower-quality games compared to engines like Unreal Engine. This reputation seems to be confirmed by [statistics](#) from the platforms Steam and itch.io: Steam has higher quality standards than sites like itch.io, which are more widely used by hobbyists and independent game developers. The statistics clearly show Unity being the engine behind the overwhelming majority of games published on itch.io, but Steam's statistics paint a very different picture: there, the most used engine is Unreal Engine, which seems to generally be considered to be the engine for higher quality games.

Unity seems to have become much more viable for high quality games: the recent High Definition Render Pipeline templates seem to have bridged the gap of visual fidelity in default scenes compared to Unreal Engine. Now, DOTS seems to be another great addition to the Unity Engine that makes it more viable to game developers or game development studios that need scalability, high performance and flexibility. Compatibility with hardware that has higher performance requirements, such as mobile and XR, is much more viable.

User friendliness

The biggest issue with DOTS in my opinion, is the user friendliness and missing features. There is much left to be done to make developing DOTS code as user-friendly as MonoBehaviours currently are. DOTS code can be a bit strange sometimes, and requires a lot of boilerplate code. Especially code for the package Unity Physics, where many systems have to be written the "old DOTS way" as there is no Physics equivalent of something like `Entities.ForEach()` or `Job.WithCode()`. I assume at some point there will be much easier ways of writing DOTS code.



DOTS Visual Scripting may be a good solution for this at some point; as a visual scripting tool might be more user-friendly to work with for those unfamiliar with DOTS syntax and might reduce the amount of boilerplate code. However, currently this package is still in the experimental phase and the developers announced in September 2020 **they will temporarily pause updates for DOTS Visual Scripting**, though they are still continuing development. There appears to be no indication for when the next versions of DOTS Visual Scripting are planned to be released; not much information about visual scripting is given, other than that it will likely be integrated into Unity's new built-in visual scripting tool, formerly known as

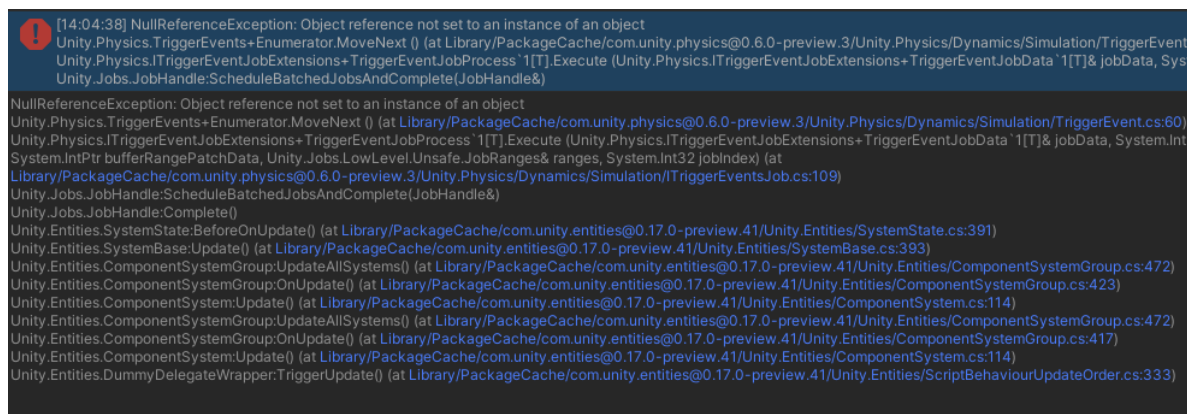
"Bolt".<https://forum.unity.com/threads/visual-scripting-roadmap-update-september-2020.978732/>

Problems found during prototyping

This chapter will describe some of the problems I personally found while trying to create a DOTS prototype using Unity version 2020 LTS with Entities 0.17 and the DOTS packages that are compatible with this version of Entities. This is only a partial list of the problems I've found, but it should give an indication of how difficult packages could be to work with at times.

Instability

Sometimes you can get error messages that do not have a stack trace to any of your code, and are not (consistently) reproducible. For example, a `NullReferenceException` occurred upon entering Play Mode:



```
[14:04:38] NullReferenceException: Object reference not set to an instance of an object
Unity.Physics.TriggerEvents+Enumerator.MoveNext() (at Library/PackageCache/com.unity.physics@0.6.0-preview.3/Unity.Physics/Dynamics/Simulation/TriggerEvent
Unity.Physics.ITriggerEventJobExtensions+TriggerEventJobProcess`1[T].Execute (Unity.Physics.ITriggerEventJobExtensions+TriggerEventJobData`1[T]& jobData, Sys
Unity.Jobs.JobHandle:ScheduleBatchedJobsAndComplete(JobHandle&)
NullReferenceException: Object reference not set to an instance of an object
Unity.Physics.TriggerEvents+Enumerator.MoveNext() (at Library/PackageCache/com.unity.physics@0.6.0-preview.3/Unity.Physics/Dynamics/Simulation/TriggerEvent.cs:60)
Unity.Physics.ITriggerEventJobExtensions+TriggerEventJobProcess`1[T].Execute (Unity.Physics.ITriggerEventJobExtensions+TriggerEventJobData`1[T]& jobData, System.Int
System.IntPtr:bufferRangePatchData, Unity.Jobs.LowLevel.Unsafe.JobRanges& ranges, System.Int32 jobIndex) (at
Library/PackageCache/com.unity.physics@0.6.0-preview.3/Unity.Physics/Dynamics/Simulation/ITriggerEventsJob.cs:109)
Unity.Jobs.JobHandle:ScheduleBatchedJobsAndComplete(JobHandle&)
Unity.Jobs.JobHandle:Complete()
Unity.Entities.SystemState:BeforeOnUpdate() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/SystemState.cs:391)
Unity.Entities.SystemBase:Update() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/SystemBase.cs:393)
Unity.Entities.ComponentSystemGroup:UpdateAllSystems() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystemGroup.cs:472)
Unity.Entities.ComponentSystemGroup:OnUpdate() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystemGroup.cs:423)
Unity.Entities.ComponentSystem:Update() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystem.cs:114)
Unity.Entities.ComponentSystemGroup:UpdateAllSystems() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystemGroup.cs:472)
Unity.Entities.ComponentSystemGroup:OnUpdate() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystemGroup.cs:417)
Unity.Entities.ComponentSystem:Update() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ComponentSystem.cs:114)
Unity.Entities.DummyDelegateWrapper:TriggerUpdate() (at Library/PackageCache/com.unity.entities@0.17.0-preview.41/Unity.Entities/ScriptBehaviourUpdateOrder.cs:333)
```

This error triggered without any user input or special conditions. Upon reading the stack trace it looked like this exception did not happen because of any changes I had made. To verify this suspicion, I immediately re-entered Play Mode without making any changes to the project at all, and this specific exception never happened again. In this case, it seems to be a bug in the physics system; while trying to create prototypes, the physics package seemed to me to be the most buggy or unreliable of all the packages I worked with.

We can only hope these kinds of issues are ironed out in packages once their status reaches "Released" status. However, because many of these bugs are not most likely not consistently reproducible it's very possible some of these can occur, even after the first verified versions of this package have been released.

Naming issues

Names in DOTS gave me more problems than I found to be reasonable. Sometimes, the names of certain DOTS classes, systems, interfaces or other components don't entirely match my expectations of what they should be. Other times, names of classes, methods, et cetera, were needlessly verbose.

ComponentDataFromEntity

For example, the type *ComponentDataFromEntity* is not a component that belongs to an entity, but rather an array of components with an entity as an indexer. So it is not a "*ComponentData*" component, it is a collection of these components. The name would indicate the syntax would look something like this:

```
ComponentDataFromEntity<CheckEntityOnGroundComponentData> data =
    GetComponentDataFromEntity<CheckEntityOnGroundComponentData>(entity);
var value = data.GroundDirection;
```

To be clear, **the code above is not a correct implementation of *ComponentDataFromEntity***. Rather, a correct implementation actually looks like this:

```
ComponentDataFromEntity<CheckEntityOnGroundComponentData> data =
    GetComponentDataFromEntity<CheckEntityOnGroundComponentData>();
var value = data[entity].GroundDirection;
```

This struct actually had a change in syntax that I personally found strange: if you want to check whether *ComponentDataFromEntity* contains a *ComponentData* associated with some entity (for those who are still confused, this means checking if an entity has a certain component), you used to be able use the method *ComponentDataFromEntity.Exists(Entity)*. However, according to the changelog for the Entities package, since Entities version 0.12.0 this has been renamed to *HasComponent(Entity)*. The same goes for the similar type, *BufferFromEntity*.

"Deprecated *ComponentDataFromEntity.Exists()*; Use *.HasComponent()* instead."

Note: *ComponentDataFromEntity<T>* is still used in a lot of tutorials to get components. However, in a *SystemBase* you can also use the method *GetComponent<T>(Entity)*, which is closer to the *MonoBehaviour* syntax and in my opinion, much easier to read.



To me, the previous name made a lot more sense. The “grammar” of DOTS code generally seems to be less sensical than the code we’re used to, and you’ll often find yourself reading right to left when trying to make sense of your code. In the example above, the question you’re asking is “Does this Entity have a component of type *CheckOnGround?*”. You would expect the code to look something like *Entity.HasComponent<CheckOnGround>()*. Rather, this code should actually be written the other way around: *CheckOnGroundComponents.HasComponent(Entity)*. If you want to read the correct intention of this code, you would have to read it right to left.

Velocity vs PhysicsVelocity

Another example is the “velocity” component in *Unity.Physics*: the actual component used to track velocity is called **PhysicsVelocity**, rather than *Velocity*. This wouldn’t be that big of an issue because Intellisense would suggest this component when you type “velocity” as well., however the type *Velocity* does already exist in the *Unity.Physics* namespace, making it much easier to accidentally use this type instead. This is not a “*IComponentData*” and is not the type of component that is added when converting a *RigidBody* or *Physics Body* component. Therefore, when iterating on a physics entity you should always use *PhysicsVelocity* instead of *Velocity*, or you will get this error:

The type 'Unity.Physics.Velocity' cannot be used as type parameter 'T' in the generic type or method 'EntityManager.GetComponentData<T>(Entity)'. There is no boxing conversion from 'Unity.Physics.Velocity' to 'Unity.Entities.IComponentData'.

Entities.WithName

One possible source of confusion that is good to be aware of is the **Entites.WithName** method: this could be confused as something that filters entities in the same way the method *.WithAll*, *.WithAny* and *.WithNone* do, but based on the name of an entity rather than a component. However, unlike these other methods, **.WithName is not an option for filtering entities**. Instead, it is a way to give your system a custom name for debugging purposes.

So where the syntax “*Entities.WithAll<PlayerTag>().ForEach*” would query only entities with the component “*PlayerTag*”, the syntax “*Entities.WithName("Player").ForEach*” **would return every single entity regardless of its name**, and instead name this system “*Player*” in the Entity Debugger and Systems windows for debugging purposes.

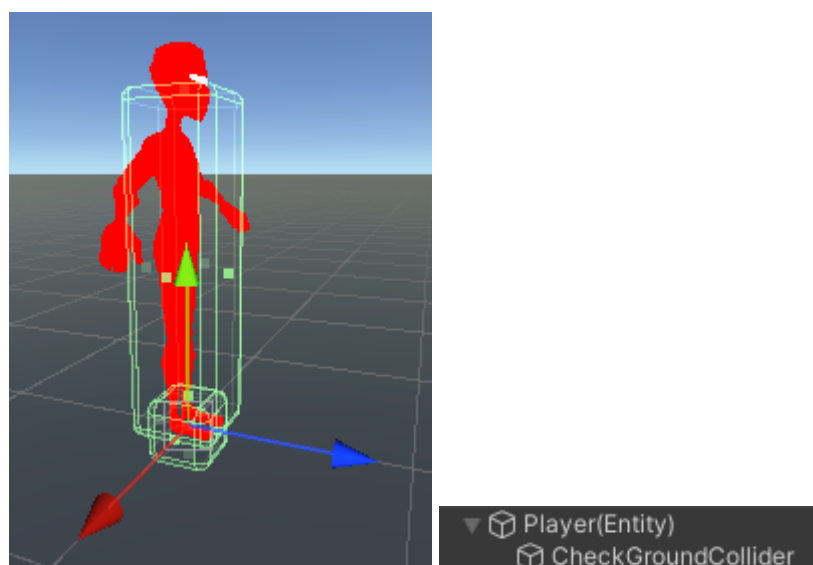
Physics triggers, collisions with child objects, compound colliders

Note that if an entity has a child entity with a collider, **the events of that child collider will reference the root entity**, rather than the entity attached to the collider itself. Rather than looking through every child entity for the component you’re looking for, which could be slow, one would be inclined to use the root object to attach any components or tags that will be used during collisions. However, giving an object a Component that is used in a “Trigger” system, seems to make the collider behave as a trigger, regardless of the collision method defined on the “*Physics Body Authoring*” component.

This means you cannot use `GetComponentData` for any components attached to the collider itself, unless you look for every child entity to try to find an object that does. So even though this behaviour is somewhat similar to the way a "Compound Collider" works in the old physics system, it is much less desirable in this scenario and takes a lot more work to implement correctly.

For example, in this context a child collider should act as a trigger to check whether this entity is on the ground. The parent object has a collider for body collisions and a physics body for movement. The "grounded" movement system, which for example enabled the ability to jump, would only run if the *OnGroundComponent* was added to an entity, which would occur on a collision between this *CheckGroundCollider* and a ground with a *GroundTag* component. The entity that needed to check whether it was or wasn't on the ground was referenced in the *GroundCollider* component.

The plan was to have the player's body collider overlap with the foot trigger in such a way that hitting a "ground" object with the side of the player would not count as being "on the ground".



This will not work as expected. Furthermore, this kind of setup could cause strange behaviour in general, especially when the parent collider has a different collision response setting on its "Physics Body" component. To see just how badly the Physics system can break if these kinds of settings, refer to the included prototype: the file `PickupTriggerAuthoring.cs` defines that struct `PickupJob` inherits from `ITriggerEventsJob`, implementing the method `Execute(TriggerEvent triggerEvent)`. Change this to inherit from `ICollisionEventsJob` instead, implementing the method `Execute(CollisionEvent triggerEvent)`, leaving the rest of the code unchanged. The physics system will completely break, making other seemingly unrelated objects' physics either stop working or act extremely inconsistent, bouncing off objects with great force or passing through them. The console will give you a lot of exceptions, none of which will at all point you in the right direction on how to fix this.

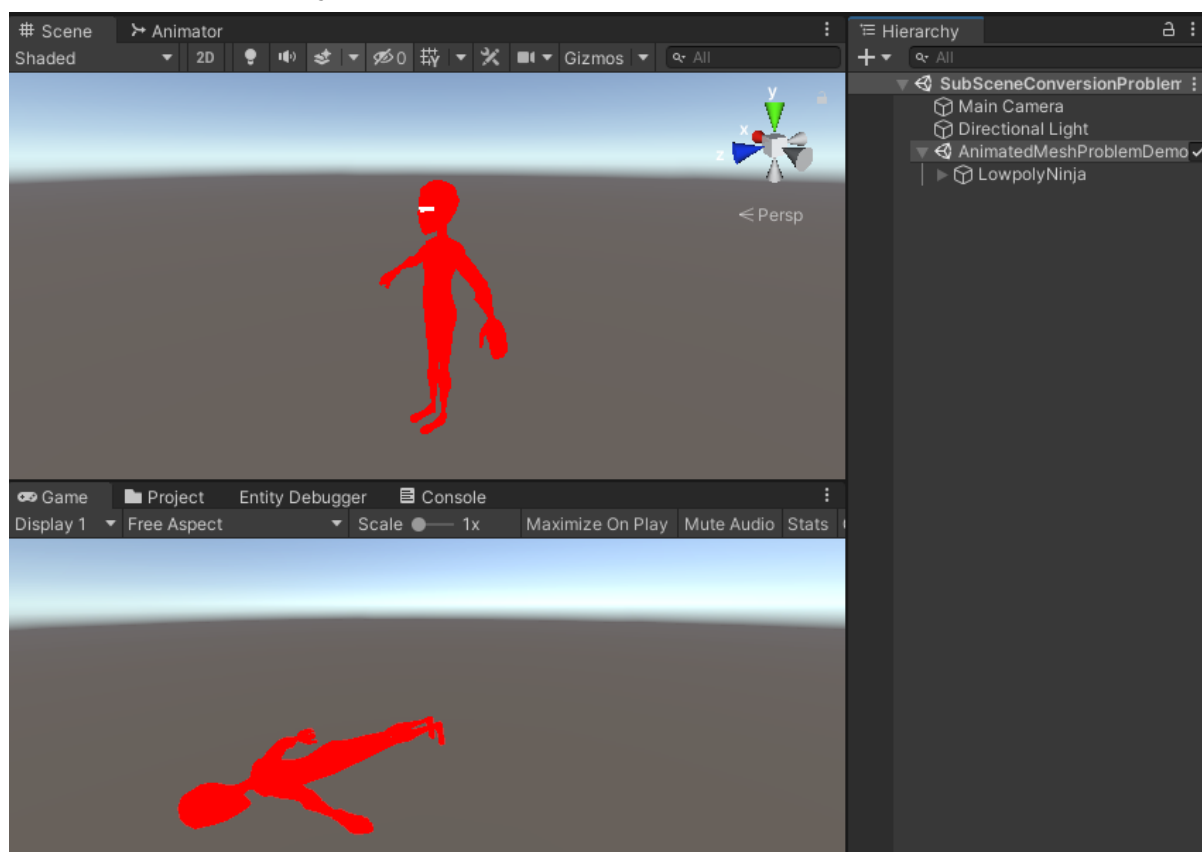
Issues with SubScenes

SubScenes are currently the only way of converting an entity before runtime. Unity recommends this as the best way to convert entities, as the component "ConvertToEntity" will be deprecated.

However, a SubScene actually works like a scene rather than a prefab. This means any Entities that are in a SubScene, can't be referenced outside of that SubScene. SubScenes, therefore, are only useful when no object in your main scene needs to directly reference any entities in this subscene.

Something unexpected happens when a SubScene is open while running it: the Scene view shows your SubScene as if it were a GameObject. In the example below, a skinned mesh renderer with an animator is being converted to an entity using a subscene. As expected, the Game view shows the model in its default pose (and in this case, an incorrect orientation) because of animated meshes not being currently supported by default in DOTS.

However, the Scene view still shows the mesh being deformed by its rig. Somehow, rather than showing the DOTS entity in the scene view, the scene view shows the (unconverted) GameObject.



As you can see, even though SubScenes are the recommended method of entity conversion, SubScenes can cause some problems and I personally avoided using them.



According to Unity's manual page on conversion, runtime conversion, such as adding a *ConvertToEntity* component in your root scene, should be a lot slower than loading a SubScene. Yet, when trying to build the application, when using a subscene the objects actually spawned into the world much later than the runtime-converted Game Objects did. So even the only mentioned advantage that could possibly be worth the issues SubScenes cause, to me does not seem to currently work as intended.

My opinion on “High Performance C#”

Even though DOTS code is technically written using C# syntax, DOTS code is very different from normal C# code. Especially when using the Burst compiler, a lot of C# functionality is gone. This is what Unity refers to as “High Performance C# (HPC#)”. In HPC#, classes cannot be used. This means that all of the concepts of Object Oriented Programming, such as inheritance and polymorphism, are no longer valid. Burst-compiled code will always operate using struct values instead.

In my opinion, the choice to make a subset of C# rather than some new programming language, comes with some undesirable side effects. Because you are still technically writing C# code, this might cause you to make incorrect assumptions about what is or isn't possible because you (and your IDE) expect to be able to do things that are valid in regular C#, such as for instance renaming variables or using field variables.

For example, one would expect to be able to declare custom variable names in a *ForEach* lambda, but this is not always the case. In this scenario, a programmer tries to access the job index of a *ForEach* lambda (which is used in cases where you would need a ‘for’ loop with an index rather than a *foreach* loop):

```
Entities.ForEach((int jobIndex, in Translation translation, in CheckEntityOnGroundComponentData checkGround) => {  
    RaycastInput input = new RaycastInput()  
    {  
        Start = translation.Value,  
        End = translation.Value + checkGround.GroundDirection,  
        Filter = collisionFilter  
    };  
    bool haveHit = collisionWorld.CastRay(input, out RaycastHit hit);  
    if(haveHit)  
    {  
        entityCommandBuffer.AddComponent<OnGroundComponentTag>(jobIndex, hit.Entity);  
    }  
}).ScheduleParallel();
```

This code will trigger the error:

“Entities.ForEach Lambda expression parameter 'jobIndex' is not a supported parameter. Supported parameter names are entityInQueryIndex,nativeThreadIndex”

This error will happen at runtime; your compiler will not warn you about this error because it is technically valid C# code.



I'm hoping the release of a fully functioning DOTS Visual Scripting tool will solve these problems; and this should also solve the issue with having to write too much boilerplate code in certain scenarios. However, DOTS Visual Scripting doesn't seem to be coming any time soon, as [Visual Scripting updates are currently paused](#). Other than that, presumably more changes would need to be made to the syntax to reduce this issue. Even with the `Entities.ForEach` and `Job.WithCode` methods that improved the process of writing and scheduling jobs, it still has some strange quirks you need to get used to, such as the lambda expression syntax needed to use these methods.

Finding correct information can also be difficult. For example, one of the first things attempted in the prototype was spawning an entity from code, using a `GameObject` reference or prefab. A lot of tutorials and examples would instruct you to just use "null" when asked for a `BlobAssetStore` instance. But in newer versions, this throws an exception.

```
GameObjectConversionSettings settings = GameObjectConversionSettings.FromWorld  
(World.DefaultGameObjectInjectionWorld, null);
```

Other things were attempted, such as creating a new `BlobAssetStore` and using that, but this would cause a warning about a memory leak. A "using" block would also result in an exception. Many answers to questions like these may go unanswered in any official Unity documentation or tutorials, and often have to be found yourself or through answers on forum threads.

Broken sample projects and outdated information

Almost all sample projects that I've tried to download were broken. Many were built on older versions of Unity and DOTS that I did not manage to get working, and after many reinstalls of different versions of all kinds of packages with little to no success, I finally gave up: I decided that from that point on that I would only work with any project that has been built using the latest version of the DOTS packages (corresponding to the release of `Entities` version 0.17). Even then, many would not work properly.

Finding correct information can also be difficult. For example, one of the first things attempted in the prototype was spawning an entity from code, using a `GameObject` reference or prefab. A lot of tutorials and examples will instruct you to just use "null" when asked for a `BlobAssetStore` instance. But in newer versions, this throws an exception.

```
GameObjectConversionSettings settings = GameObjectConversionSettings.FromWorld  
(World.DefaultGameObjectInjectionWorld, null);
```

Other things were attempted, such as creating a new `BlobAssetStore` and using that, but this would cause a warning about a memory leak. A "using" block would also result in an exception. Many answers to questions like these may go unanswered in any official Unity documentation or tutorials, and often have to be found yourself or through answers on forum threads.



Things I am currently satisfied with, or believe can't be improved upon

Writing ECS code

Writing systems using **Entities.ForEach** seems to be user friendly enough to be able to write simple code with the same speed as you would write the same code in a MonoBehaviour. The lambda structure is strange at first but it doesn't seem to matter much once you're used to it. Even though I'm still not a great fan of the exact way this has been implemented, it's a huge improvement compared to earlier versions of Entities, which needed to define your Jobs, EntityQueries, et cetera separately, rather than all those things being automatically generated by the much more user-friendly Entities.ForEach.

According to the survey many users don't think writing ECS code takes longer, and all respondents seem to prefer ECS/DOD code over OOP/classic Unity. The potential gain in performance and other benefits gained by using DOTS systems correctly can often make the trouble worth it.

Even though many developers, such as some respondents to the DOTS Survey(Attachment 1), are critical of Unity's radio silence, and feel that much needs to be improved in many aspects of DOTS, all of them are still very excited about DOTS. DOTS enables developers to think bigger. Less restrictions in performance enable different game designs, or can compensate for the slower performance of certain hardware like mobile and VR. DOTS is, in many cases, possibly the best solution a developer might have to their performance problems.

Sometimes it can even make programming easier, as you no longer really have to think much about perfectly optimizing code because the performance gains from using DOTS is already so large, it can sometimes easily compensate for worse code. Mike Geig shows this in his [survival shooter example](#), where he manages to easily run more than 60 frames per second with around 2.100 enemies and over 50.000 bullets, with what he calls "probably the worst piece of code I've ever written"(39:48-42:08):

I also want to point out [...] the collision system in this is awful. It's probably the worst piece of code I've ever written. [...] I wrote it this way intentionally to show you that basically, what's happening is every enemy is gonna check for collision against every single bullet, even if they're nowhere near each other, and then the player is gonna check for collision against every single enemy, every single frame on the main thread. It's beautiful how awful this is. Don't write your code like that, but I wanted to show you the performance we're gonna get out of this, [...] even though it's so, so, so awful.

The Job System and Burst Compiler, both marked as "Release" status in the package manager, seem to be very usable, bug free and stable. Once you are used to the restrictions of the Burst Compiler, it becomes very clear when you should or should not use it. The Job System and Burst Compiler have complete documentation and should be supported by the newest versions of the Unity Editor.

File structure

If a system only needs to define a single authoring component, you can combine several aspects of a functionality (including the authoring component, data components, and systems) can all be defined in the same class file. It would be nice to be able to define multiple authoring components in a single file, but this is already a restriction for MonoBehaviour components rather than one caused by any DOTS systems.

It is especially nice to be able to write a single authoring component to add multiple runtime components. That makes it easier to, for example, define a "type" (in this case, part of the archetype) through adding a single authoring component, rather than having to remember the correct combinations of components used by your systems.

Things I feel need improvements or am unsatisfied with.

DOTS in its current state, is not very user friendly and **I would personally not use it in rapid prototyping or time-sensitive production**. This is a list of things I would personally need to see in future releases before I would reconsider using DOTS as a prototyping tool or for more time-sensitive production. To reiterate, this is all based more on personal opinion than fact and this does not represent the opinions of Rebels or anyone other than the author. Note that this list was written about the DOTS packages released with Entities version 0.17.

- Being able to select and move Entities in the scene view the same way you could with GameObjects, and being able to edit the component data in the "Entities" window (or any other window that allows you to inspect component data).
- Writing less "boilerplate code"; especially a new syntax for collisions that comes a bit closer to the MonoBehaviour methods such as *OnCollisionEnter* and *OnTriggerStay*.
- Not having to create a new C# file for every Tag or ComponentData. If data should be split up into multiple separate structs to process data more efficiently, this shouldn't come at the cost of increased number of files or folder structure complexity. Creating empty structs to define a Tag already feels redundant, having to do this in a separate file seems even more redundant. One way to reduce the amount of files is to, whenever possible, create a single authoring component that attaches multiple data components to an entity and define those data components in the same file. The restriction to have one authoring component/object component per class was already a restriction for MonoBehaviours, but this was less limiting as MonoBehaviours are usually much larger than DataComponent structs, and defining a completely empty class or struct was never necessary for MonoBehaviours.
- Being able to reference entities that are converted before runtime. In the current version of Entities, version 0.17, the only way to convert entities before runtime is by using SubScenes. However, this causes references from outside of that subspace to break because of a "scene mismatch". Once prefabs or non-SubScene GameObjects can be converted before runtime, this issue should be resolved.

- The physics system caused me the most trouble. The most horrible bugs I have ever seen in my life, such as the entire physics world apparently breaking when trying to spawn a new physics object.
- More production-ready features/packages, such as the rendering, animation and physics, so that some kind of work around or hybrid solution is not needed. Unity Physics in particular was very hard to work with because of a lot of bugs, missing features and amount of needed boilerplate code for even simple functions like triggers or collisions.
- Most importantly, we need proper documentation, tutorials and samples. Finding a sample that actually worked without having to fix errors, even in some projects released for the exact same version of the package Entities and/or the same Unity version, was surprisingly rare. Even those that ran without any build errors usually had some sort of issue at runtime. This was an issue for almost **any** project by Unity Technologies, and most projects by third parties. Outdated tutorials could also cause frustration when trying to use old systems or syntax.

DOs and DON'Ts

This is a list of DOs and DON'Ts based on the personal experience of and research by the author and therefore **these statements do not necessarily represent the advice or opinions of anyone but the author.**

DON'T use Unity Editor version 2021 with any DOTS packages other than those marked as "Released" (e.g., Jobs, Burst), unless you have verified that the package versions you're using are compatible by, for example, reading the DOTS forum to find threads about new updates. Currently the thread warning about version 2021's incompatibility with DOTS is a sticky post on the DOTS forum; as long as this is the case, you can assume the warning about incompatibility is still accurate.

DON'T upgrade every package to the newest version without first checking compatibility with other DOTS packages and making a backup of your project. For example, Burst can currently be upgraded to versions 1.4.8, which is incompatible with the most recent versions of other DOTS packages such as Entities 0.17.

DO become familiar with the concepts of Data Oriented Design before you use the Entities package. Trying to fit OOP styles of programming into ECS will cause a lot of trouble. You can, however, use classic Unity code together with ECS using a hybrid implementation if an OOP style is preferred for some functionality.

DO familiarize yourself with the Job System and Burst, and use them whenever you can. Job System and Burst are both no longer considered "Experimental" or "Preview"/"Pre-Release" packages, and should be compatible with all new Unity versions. Even when not planning to use any DOTS package in your project currently, try to build your code in a way that could easily be converted to be compatible with Burst/Jobs.

DON'T read old manuals/documentation. Always ensure that you are reading the documentation for the correct package versions, as a lot has changed since the first versions of DOTS packages. This can be an issue when using search engines to find information.

DON'T install every DOTS package separately. Instead, start with the packages that have the most amount of dependencies on other DOTS packages to ensure the versions are compatible; as it will automatically install the correct versions of its dependencies. For example, you can install the Hybrid renderer to install Jobs, Burst and Entities as well.

DON'T use SubScenes. This point might be one of the more controversial opinions as this directly contradicts Unity's official advice about conversion. However, my experience with subscenes has not been great. Even though subscenes are supposed to convert much more quickly than GameObjects in your root scene with a *ConvertToEntity* component, it seems that these GameObjects actually seem to convert faster in a build than these SubScenes. In the prototype, for example, I tried to put the environment into a subscene while keeping the player entity as a GameObject with *ConvertToEntity*. The player actually fell through where the floor should have already been loaded, and could either bounce very high into the air or even fall all the way through the floor and bottom kill plane.

Common issues

These are some issues I ran into during the prototyping phase. It often took me a lot of time to debug, as error messages often don't really seem to point you in the right direction. That's why I decided to include this chapter: being aware of these issues might prevent you from wasting time on debugging these errors in the future.

My entities are invisible or pink

First, **make sure the Hybrid Renderer is installed**. You might have some issues with the hybrid renderer depending on what render pipeline you use. During prototyping, I personally used **Hybrid Renderer V2** with the **Universal Render Pipeline(URP)** in Unity 2020 LTS; this is a newer version of the Hybrid Renderer; According to the [manual for the Hybrid Renderer](#), Hybrid Renderer v1 will be deprecated and currently already has less features than v2, such as error messages. Also note that the default renderer is **not supported by v2 (refer to the paragraph: "Installing packages" on how to install the Hybrid Renderer and enable v2)**.

The default URP material does not seem to work. In some cases, materials will show up as an unlit pink color (the default color for missing materials or shader errors). In other cases, materials might be completely invisible. If the color is pink, check to see if your project materials need to be upgraded to a URP compatible material; if the material you're using was meant for the default renderer, URP/HDRP will show this material as pink. If that doesn't work and/or your entity is completely invisible, try to make a new material from an empty shader graph, and use this material instead.

Of course, if your entity is invisible, you should also check you actually set up the components correctly. A normal mesh renderer should always convert to their DOTs counterparts, such as the component RenderMesh. Use the Entity Debugger or Entities window to verify this component is added to your entity. If not, something has gone wrong during conversion.

The samples provided by Unity do not work

Try to find examples created by third parties: Unity's samples are currently very outdated, and trying to make them work usually resulted in more time wasted on debugging and attempting to solve the issues, reinstalling different versions of packages, the Unity Editor, render pipelines, et cetera. Try to look for projects made with the most recent versions of DOTs packages to prevent incompatibility issues.



The physics are broken

Physics seemed to be the most problematic package I tried working with. As some of its functionality (such as detecting triggers/collisions) are very important for most games, including the design of the prototype, it was pretty important to try to implement this package. However, it can be extremely unstable. For example, in the [prototype](#) you can change two lines of code to seemingly break the entire game; changing the struct *PickupJob* in *Pickups/PickupTriggerAuthoring.cs* to implement the *ITriggerEventsJob* interface instead of the *ICollisionEventsJob* interface, and change the parameter in the method *Execute* to use a *TriggerEvent* instead of a *CollisionEvent*. By entering Play Mode after saving these changes, you will most likely immediately notice the issues this can cause.

A method I need is missing

One of the largest drawbacks of DOTS currently is that a lot of features are either still missing or difficult to find because of the differences in naming. For example, because the *Transform* class does not exist in DOTS you might have difficulty finding methods similar to the commonly used *Transform.MoveTowards(Vector3 position)* or *Transform.RotateAround(Vector3 point, Vector3 axis, float angle)*. These kinds of things are a big part of what makes DOTS feel almost like learning a new engine rather than just using some package; there are a lot of new things to get used to or things that need to be relearned.

I can't reference an entity in the inspector

Though the presentation "[Options for Entity interaction](#)" provides great insight on how to think about entity/data relationships, it never actually demonstrates how to set up these relationships through the editor like you would with *GameObjects*. What I found is that you can reference an Entity (meaning, a *GameObject* that is to be converted to an entity at runtime) when using the tag *[GenerateAuthoringComponent]*. In this case, your Data Component would simply have a field of the type *Entity* and it will serialize in the inspector as expected. However, sometimes you want to set up a custom authoring component. In this case, setting up a reference is much harder, as simply having an "Entity" field will not be serialized like it would with a generated authoring component. This also makes it difficult to reference an entity in a *GameObject* for a hybrid solution. The attached [prototype](#) contains a couple of hybrid implementations that get around this restriction, such as the *EntityReferenceReceiver* and *EntityReferenceSender* components: these will set up a reference to an entity on a separate component.



Important sources

This chapter contains a list of sources that, in my personal experience, have been helpful in learning DOTS, or are often referred to in this document. A complete list of sources including author and date are included in the chapter [Source List](#); this chapter serves instead as a quick reference for any Unity developer exploring DOTS systems.

Sources from Unity Technologies

The official Unity website contains an [introduction to DOTS](#), and [an overview of every package currently considered to be part of DOTS](#) with its current development status. It also contains links to the latest manual pages.

Unity Learn, which is currently free for all users, has some courses about DOTS:

- "[Entity Component System](#)" is a compilation of video tutorials which introduce the core concepts of DOTS: the **Entity Component System** structure, the **Job System**, and the **Burst Compiler**.
- "[What is DOTS and why is it important?](#)" is a more detailed explanation of DOTS and the concepts of **multithreading** and **Data-Oriented Design**.
- "[DOTS Best Practices](#)" is an in-depth guide to the best practices DOTS developers should be aware of. It consists of three parts:
 1. Understand data-oriented design
 2. Data design
 3. Implementation and optimization

[C# Job System tips and troubleshooting](#) is a page on common problems people will run into when starting to learn to use the Job System, and how to fix or avoid them.

[Common patterns in gameplay code](#) is a page about a few patterns you can use when implementing the Entities package, such as using static methods in an `Entities.ForEach` query.

Community sources

[Unity's DOTS forum](#) is a place where users discuss DOTS packages. This forum is mostly about general DOTS info or about the core DOTS packages such as Entities, Jobs and Burst. There are more specialized forums about specific DOTS packages linked at the top of this forum, such as the Physics forums. These forums are often used by Unity's DOTS team to share updates and news regarding DOTS packages, or to discuss design choices and clarify the philosophy behind DOTS systems.

There are also a number of content creators on Youtube who have made tutorials about DOTS, however the community is still very small. These tutorials are often outdated when discussing specific code syntax, as this has changed much in recent versions of DOTS packages.

The channel "[Turbo Makes Games](#)" is mostly centered around game development with DOTS, hosted by game developer Johnny Thompson. His channel is currently the biggest source of DOTS tutorials, especially for up to date tutorials using the latest versions of DOTS packages. He also regularly hosts live shows and interviews where he talks to other game developers, often discussing their experience with DOTS and the problems they've run into.. Other examples of channels that have uploaded DOTS-related content are "[Dapper Dino](#)" and "[Code Monkey](#)", though their videos are usually more focused on game development in Unity in general rather than DOTS/ECS.

Glossary

At least some level of knowledge about C# and Unity is expected from the reader. Therefore, this glossary will not necessarily include common C# or programming specific terms such as "struct", "class", "lambda", et cetera. If any term in this advisory report is unfamiliar and not present in this list, some independent research (e.g. Google/DuckDuckGo, Microsoft Docs) may be necessary.

Authoring - creating components in Unity's scene view. In ECS, authoring components use a `GameObject` to define

Experimental package - see "**Released package**"

Hybrid DOTS - refers to using DOTS and MonoBehaviours in the same project or on the same object. Oftentimes this is done to implement DOTS into an already existing project without having to rewrite everything, or when a certain feature does not have a DOTS implementation. This term was more popular in earlier stages of DOTS and nowadays is considered by some to be somewhat obsolete, because the way of creating hybrid versus "pure" DOTS/ECS used to have more differences. However, it is still used in Unity's API, like the method `AddHybridComponent(UnityEngine.Component)` which adds a MonoBehaviour component to an Entity.

Managed - e.g. "**managed components**" or "**managed objects**": Refers to objects that are managed by the Garbage Collector. Therefore, these do not need to be manually disposed through code. In the context of DOTS this often refers to MonoBehaviour classes, where "unmanaged (components)" usually refers to `IComponentData` structs.

Package Manager - Unity's built-in tool for managing and importing project packages, package versions, and assets bought through Unity's Asset Store.

Preview package - see "**Released package**"

Pre-release package - see "**Released package**"

Released package - refers to Unity Packages that are included in Unity's Package Manager. Unity packages have three different states of progress¹:

- **Released** - Previously known as "**Verified**" packages, these packages are visible by default in the Package Manager and are usually well-documented and feature complete.
- **Pre-release** - previously known as "**Preview**" packages, these packages are supported and are usually feature-complete and stable enough, though not officially recommended, for production. These packages are not visible in the Package Manager by default, but can be enabled in your Project settings (As of Unity version 2021 LTS, this setting can be found in *Edit > Project Settings... > Package Manager > Advanced Settings > Enable Preview Packages*. Any future versions of **Released** packages that are available for testing will be considered a Pre-Release package before it becomes the released version..

¹ <https://unity.com/experimental-and-pre-release-packages>

- **Experimental** - Experimental packages are packages that are not visible in the Package Manager. These packages often lack documentation, and though often already available to the public for testing, are not guaranteed to ever become a pre-release version and should not be considered stable enough for production.

Structural change - a change that modifies the "structure" of your data: adding/removing entities or components or changing the value of a SharedComponentData. See paragraph: "Sync Points and Structural Changes"

Verified package - see "Released package"

Source list

Acton, M. [Mike Acton]. (2017, March 24). CppCon 2014 Mike Acton Data Oriented Design and C++ [Video]. YouTube. <https://www.youtube.com/watch?v=g2KFSD3ObrY>

Acton, M. [unitytechnologies]. (2020, August 14). Unity R&D team 2021 Roadmap AMA [Forum post]. Reddit. https://www.reddit.com/r/Unity3D/comments/igmeq1/unity_rd_team_2021_roadmap_ama/g1ghn8i/

aksyr. (2020, February 5). Unity DSP Graph Modular Synth [Software]. UnityList. <https://unitylist.com/p/ut9/Unity-DSP-Graph-Modular-Synth>

Android Developers. (n.d.). Instant play games |. Retrieved March 7, 2021, from <https://developer.android.com/topic/google-play-instant/instant-play-games>

ans_unity. (2020, August 18). Unity - The road to 2021 - Q&A [Forum post]. Unity Forum. <https://forum.unity.com/threads/the-road-to-2021-q-a.950158/page-2#post-6219669>

Ante, J. [Joachim_Ante]. (2019, October 3). Using Unity Terrain with DOTS workflow [Forum post]. Unity Forum. <https://forum.unity.com/threads/using-unity-terrain-with-dots-workflow.755105/#post-5028386>

Ante, J. [Joachim_Ante]. (2020a, March 28). Unity - Hybrid Renderer V2 (0.4.0) [Forum post]. Unity Forum. <https://forum.unity.com/threads/hybrid-renderer-v2-0-4-0.847546/#post-5641513>

Ante, J. [Joachim_Ante]. (2020b, October 4). Looking forward to DOTS roadmap information in 2020 [Forum post]. Unity Forum. <https://forum.unity.com/threads/looking-forward-to-dots-roadmap-information-in-2020.980850/#post-6382230>

Ante, J. [Unity]. (2017, July 10). Unite Europe 2017 - C# job system & compiler [Video]. YouTube. <https://www.youtube.com/watch?v=AXUvnk7Jws4&feature=youtu.be>

Antypodish. (2021, February 21). [Open Source] Genetic Neural Network - F1Cars (Feedback wanted) [Forum post]. Unity Forum. <https://forum.unity.com/threads/open-source-genetic-neural-network-f1cars-feedback-wanted.1061879/>

Baumel, E. [Unity]. (2019, April 9). Understanding data-oriented design for entity component systems - Unity at GDC 2019 [Video]. YouTube. https://www.youtube.com/watch?v=0_Byw9UMn9g

Blondin, S. (2021, March 22). A new Package Manager experience in Unity 2021.1. Unity Blog. <https://blog.unity.com/technology/a-new-package-manager-experience-in-unity-2021>

Cambridge Dictionary. (n.d.-a). Community. In Cambridge Business English Dictionary. Retrieved July 7, 2021, from <https://dictionary.cambridge.org/dictionary/english/community>

Cambridge Dictionary. (n.d.-b). Performance. In Cambridge Business English Dictionary. Retrieved July 7, 2021, from <https://dictionary.cambridge.org/dictionary/english/performance>

Clock cycle. (2010). In TechTerms. <https://techterms.com/definition/clockcycle>

Dis/Assembling CIL Code | Mono. (n.d.). Mono. Retrieved July 7, 2021, from <https://www.mono-project.com/docs/tools+libraries/tools/monodis/>

dongyiqi. (2018, April 13). How to implement FSM in ECS architecture [Forum post]. Unity Forum. <https://forum.unity.com/threads/how-to-implement-fsm-in-ecs-architecture.526567/>

Dupuis, B. [benoitd_unity]. (2020, September 30). Unity - UI Roadmap - Q3 2020 [Forum post]. Unity Forum. <https://forum.unity.com/threads/ui-roadmap-q3-2020.980202/>

Dutton, F. (2012, April 18). What is Indie? Eurogamer.Net. <https://www.eurogamer.net/articles/2012-04-16-what-is-indie>

Eliasson, S. (2019, October 7). Creating a third-person zombie shooter with DOTS - Unite Copenhagen. YouTube. <https://www.youtube.com/watch?v=yTGhg905SCs>

Fabian, R. (2013, June 25). Finite State Machines. Data-Oriented Design. <https://www.dataorienteddesign.com/dodmain/node8.html>

Ford, T. [GDC]. (2019, February 8). Overwatch Gameplay Architecture and Netcode [Video]. YouTube. <https://www.youtube.com/watch?v=W3aieHjyNvw>

Geig, M. [Unity]. (2019a, September 26). Converting your game to DOTS - Unite Copenhagen [Video]. YouTube. <https://www.youtube.com/watch?v=BNMrevfB6Qo>

Geig, M. [Unity]. (2019b, December 20). Getting started with DOTS: Scripting Pong (Tutorial) [Video]. YouTube. <https://www.youtube.com/watch?v=agAUXNFBWt4>

Goldstone, W. [Unity]. (2019, September 26). What to expect in 2020: Unity roadmap - Unite Copenhagen 2019 [Video]. YouTube. <https://www.youtube.com/watch?v=hRJUt3voDHQ>

Goldstone, W. [Unity]. (2020, March 25). Unity Roadmap 2020: Core Engine & Creator Tools [Video]. YouTube. <https://www.youtube.com/watch?v=dDjsS4NPqFU>

Hammerton, L. [Unity]. (2019, October 9). Getting started with Burst - Unite Copenhagen [Video]. YouTube. <https://www.youtube.com/watch?v=Tzn-nXghK1o>

Havok. (n.d.). About Havok. Retrieved July 7, 2021, from <https://www.havok.com/about-havok/>

Havok. (2019, September 17). Havok Physics for Unity : Stacking Stability [Video]. YouTube. <https://www.youtube.com/watch?v=ctgymidOWPo>

Johansson, T. [GDC]. (2018, March 30). Unity at GDC - Job System & Entity Component System [Video]. YouTube. <https://www.youtube.com/watch?v=kwnbgClh2ls>

Johnson, W. [Unity]. (2018, November 21). ECS Track: Graph Driven Audio in an ECS World - Unite LA [Video]. YouTube. <https://www.youtube.com/watch?v=kDE-KxQBiyQ>

- Kumar, R. (2020, March 5). What are Threads in Computer Processor or CPU? GeeksforGeeks.
<https://www.geeksforgeeks.org/what-are-threads-in-computer-processor-or-cpu/>
- LeonhardP. (2020, August 12). Unity - The road to 2021 - Q&A [Forum post]. Unity Forum.
<https://forum.unity.com/threads/the-road-to-2021-q-a.950158/>
- MartinGram. (2021, April 12). Unity - Notice on DOTS compatibility with Unity 2021.1 [Forum post]. Unity Forum.
<https://forum.unity.com/threads/notice-on-dots-compatibility-with-unity-2021-1.1091800/>
- Meijer, L. (2019a, February 26). On DOTS: C++ & C#. Unity Blog.
<https://blog.unity.com/technology/on-dots-c-c>
- Meijer, L. (2019b, March 8). On DOTS: Entity Component System. Unity Blog.
<https://blog.unity.com/technology/on-dots-entity-component-system>
- Merkus, J. (2021, June 14). Voorbeeld APA-stijl: TED-talk. Scribbr.
<https://www.scribbr.nl/apa-voorbeelden/ted-talk/>
- Methods - ICT research methods. (2021, July 5). ICT Research Methods.
<https://ictresearchmethods.nl/Methods>
- Microsoft. (2017, March 30). Blittable and Non-Blittable Types. Microsoft Docs.
<https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types?redirectedfrom=MSDN>
- Mutel, A. (2021, April 14). Bursting into 2021 with Burst 1.5. Unity Blog.
<https://blog.unity.com/technology/bursting-into-2021-with-burst-15>
- Rebels. (n.d.). Philosophy. Retrieved March 7, 2021, from <https://www.rebels.io/philosophy>
- Santema, S. [Unity]. (2019, October 14). Extending the Animation Rigging package with C# - Unite Copenhagen [Video]. YouTube. <https://www.youtube.com/watch?v=xwwKolAlSyY>
- Scott, A. (2020, May). DOTS Runtime - Unity Platform - Rendering & Visual Effects. Product Roadmap.
<https://portal.productboard.com/unity/1-unity-platform-rendering-visual-effects/c/128-dots-runtime>
- simonbz. (2020, July 7). Can I use Animation Rigging with DOTS? [Forum post]. Unity Forum.
<https://forum.unity.com/threads/can-i-use-animation-rigging-with-dots.926321/#post-6064136>
- Tak. (2020, February 14). Unity - DOTS Audio Discussion [Forum post]. Unity Forum.
<https://forum.unity.com/threads/dots-audio-discussion.651982/page-4#post-5480133>
- Thompson, J. [Turbo Makes Games]. (2021, March 12). "Hybrid ECS" Does NOT Exist - Unity DOTS 2021 [Video]. YouTube. <https://www.youtube.com/watch?v=AgdVM01-W5w>

uDamian. (2021, January 27). Unity - DOTS Editor 0.12.0-preview Release: The new way to inspect Entities [Forum post]. Unity Forum.

<https://forum.unity.com/threads/dots-editor-0-12-0-preview-release-the-new-way-to-inspect-entities.1045306/#post-6772715>

Unity. (2020, June 24). Stacking up the new functionality in DOTS Physics | Unite Now 2020 [Video]. YouTube. https://www.youtube.com/watch?v=n_5RGdF7Doo

Unity. (2021, March 27). Unity GDC Showcase 2021 [Video]. YouTube. <https://www.youtube.com/watch?v=na7EMenl2lY>

Unity Technologies. (n.d.-a). C#/.NET Language Support | Burst | 1.5.4. Unity. Retrieved July 7, 2021, from https://docs.unity3d.com/Packages/com.unity.burst@1.5/manual/docs/CSharpLanguageSupport_Types.html

Unity Technologies. (n.d.-b). Conversion Workflow | Entities | 0.17.0-preview.42. Unity. Retrieved July 7, 2021, from <https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/conversion.html>

Unity Technologies. (n.d.-c). Creating gameplay | Entities | 0.17.0-preview.42. Unity. Retrieved July 7, 2021, from https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/gp_overview.html

Unity Technologies. (n.d.-d). Design philosophy | Unity Physics | 0.6.0-preview.3. Unity. Retrieved July 7, 2021, from <https://docs.unity3d.com/Packages/com.unity.physics@0.6/manual/design.html>

Unity Technologies. (n.d.-e). DOTS - Unity's new multithreaded Data-Oriented Technology Stack. Unity. Retrieved March 7, 2021, from <https://unity.com/dots>

Unity Technologies. (n.d.-f). DOTS packages. Unity. Retrieved July 7, 2021, from <https://unity.com/dots/packages>

Unity Technologies. (n.d.-g). Fixing Performance Problems. Unity Learn. Retrieved July 7, 2021, from <https://learn.unity.com/tutorial/fixing-performance-problems>

Unity Technologies. (n.d.-h). Modding Support | Burst | 1.5.4. Unity. Retrieved July 7, 2021, from <https://docs.unity3d.com/Packages/com.unity.burst@1.5/manual/docs/ModdingSupport.html>

Unity Technologies. (n.d.-i). Unity - Manual: Important Classes. Unity. Retrieved March 7, 2021, from <https://docs.unity3d.com/Manual/ScriptingImportantClasses.html>

Unity Technologies. (n.d.-j). What is DOTS and why is it important? Unity Learn. Retrieved July 7, 2021, from <https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important>

Unity Technologies. (2017, November 8). We're joining Unity to help democratize data-oriented programming. Unity Blog. <https://blog.unity.com/community/were-joining-unity-to-help-democratize-data-oriented-programming>

Unity Technologies. (2018a, June 15). Unity - Manual: NativeContainer. Unity.
<https://docs.unity3d.com/Manual/JobSystemNativeContainer.html>

Unity Technologies. (2018b, June 15). Unity - Manual: The safety system in the C# Job System. Unity. <https://docs.unity3d.com/Manual/JobSystemSafetySystem.html>

Unity Technologies. (2020a, June 2). Unity - Manual: Understanding Automatic Memory Management. Unity.
<https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html>

Unity Technologies. (2020b, July 20). Package Manager updates in Unity 2020.1. Unity Technologies Blog.
<https://blogs.unity3d.com/2020/06/24/package-manager-updates-in-unity-2020-1/>

Attachment 1: DOTS Survey

The following survey has been posted to Unity's DOTS forum in May 2021, in an attempt to gain insight into the way DOTS users see and use DOTS, and to what extent DOTS packages are considered to be usable for production.. There were 23 total responses.

A few biases must be addressed. First of all, the assumption was made that writing DOTS code would likely take programmers longer than classic Unity. However, the amount of people reported not spending any extra time on DOTS code was much larger than initially expected. Had this been foreseen, questions would have been asked about whether DOTS code could actually be written faster in general, as many also report generally preferring ECS/Data-Oriented Design over Object Oriented programming. Possibly, to some users, developing with DOTS may not be only just as fast as classic Unity, but may be even faster.

Another question that would have been helpful is about the respondents' general programming and/or game development experience. For instance, one respondent reported themselves to be a beginner when it comes to experience with Unity, but also mentions having almost three decades of experience with programming in general. This experience, though significant, is not reflected in the statistics because of the lack of this question.

We also need to keep in mind that although 23 responses is more than expected, a sample size of 23 is still very small, and therefore no definite conclusions can be drawn from this survey. More importantly, any data gathered will reflect only the opinion of DOTS forum users, so any opinions would likely be skewed towards a preference for DOTS. It seems reasonable to assume that posting the same survey on another day or another month could have yielded vastly different results, and most definitely posting the survey to a forum that is not specifically for DOTS users would have had an even bigger impact on the results.

However, some of the results clearly show patterns which can indicate some sort of general consensus amongst DOTS users on specific topics, such as whether the community considers specific packages to be usable for production.

This survey has four distinct categories:

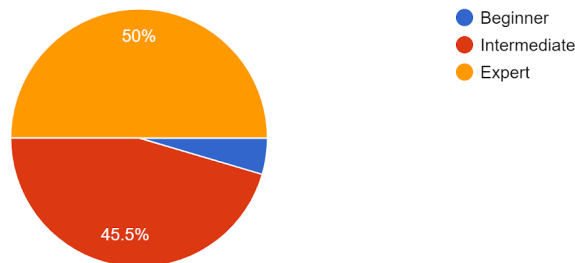
- Questions about the respondents' level of experience with Unity and DOTS packages
- Questions about the respondents' knowledge about DOTS concepts before working with DOTS packages
- Questions about specific packages and the usability of, or experience with, these packages.
- Final notes, where respondents were able to leave comments about their experience with DOTS or give advice to a developer if they were to just get started using DOTS packages.



About the respondents' level of experience, and their teams

What is your level of experience with Unity?

22 responses

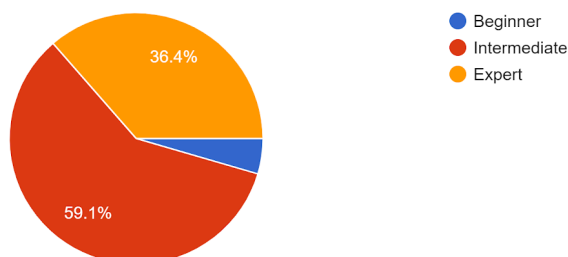


Only one respondent reported themselves to be a beginner with Unity, however, this person does have 27 years of experience developing with Object Oriented Programming, and reported to be very familiar with the concepts of Data-Oriented Design. Others are evenly split between intermediate and expert.

Keep in mind this was posted to Unity's DOTS forum. People who frequently visit the forum page of an experimental Unity feature may tend to be more experienced with Unity in general. However, though it might be a contributing factor, this might not necessarily explain the large number of people who consider themselves to be an expert relative to the amount of intermediate-level developers. Another possible explanation could be that DOTS is unattractive to beginners, which is why only intermediate/expert level Unity developers use DOTS regularly. This would correlate with the recommended level of experience on Unity Learn's DOTS courses (such as [DOTS best practices](#)).

What do you consider to be your level of experience with DOTS packages?

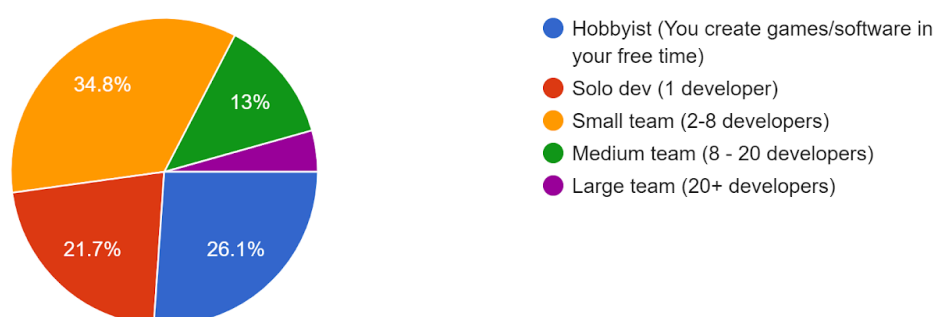
22 responses



Though the amount of people who consider themselves to be a beginner with DOTS packages is surprisingly low, this again could be explained by the fact that respondents found this thread on the DOTS forum. While the majority (59%) considers themselves to be intermediate, a large number of people still consider themselves an expert (36%).

What would you consider the most accurate description of yourself or your team?

23 responses



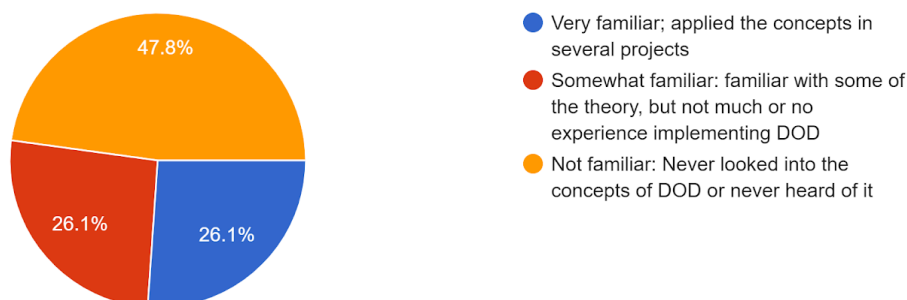
Interestingly, more than a quarter of DOTS forum users describe themselves as a hobbyist. Another 22% are solo developers, meaning only around 50% of DOTS forum users work in a team professionally. The majority of respondents work in a small team of 2 to 8 developers (35%), and only a single person reports working in a large team of more than 20 developers.



Knowledge about DOTS concepts (Data-Oriented Design, Multithreading) before using DOTS

Before you started using Entities, how familiar were you with Data-Oriented Design/Programming(DOD)?

23 responses



Before you started using Jobs, how familiar were you with multithreading? (Note: async and coroutines are not multithreading)

23 responses



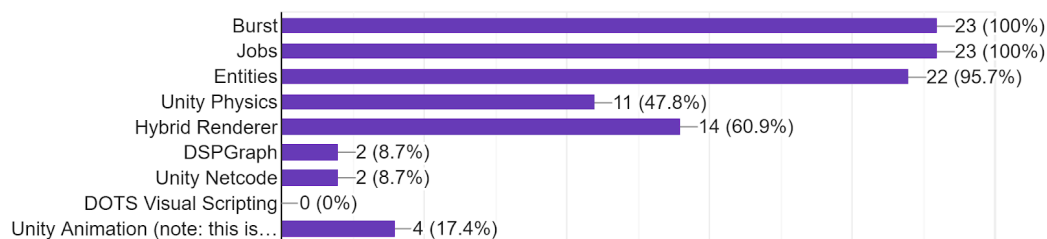
It is clear that multithreading is a more popular topic than DOD. A total of 48% of users is unfamiliar with DOD, while only 4% is unfamiliar with multithreading. However, of the people that are at least somewhat familiar with the concepts, a larger percentage of this group considers themselves very familiar to DOD, compared to multithreading. One can only speculate about why this is; however we can assume that because the survey only has 23 responses, it seems likely there is no clear reason for this discrepancy and this might not have occurred if there was a larger response.



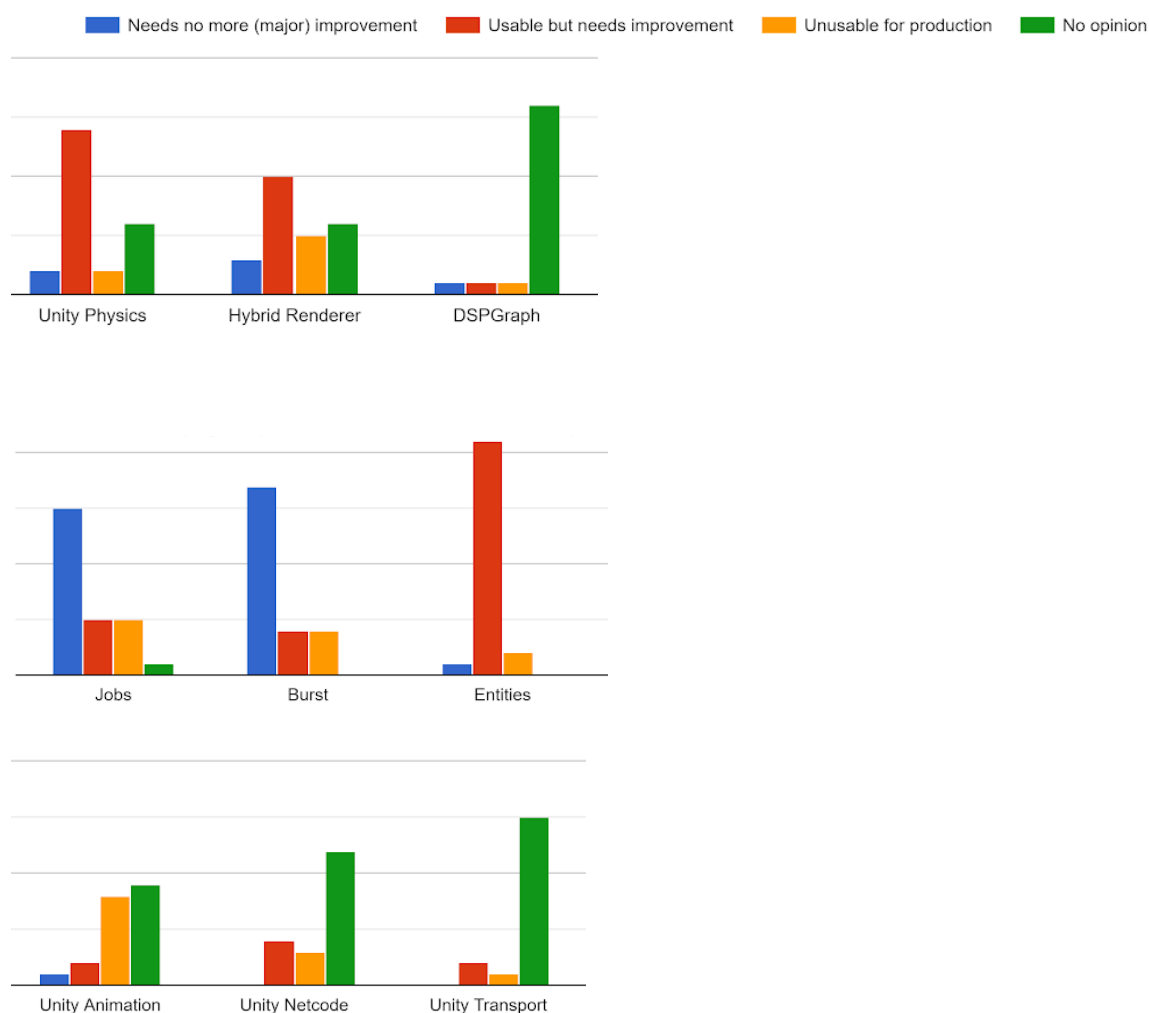
Package usage and usability

Which DOTS packages do you currently use regularly?

23 responses



In your experience, which of these DOTS packages would you consider to be good enough for production?



There seems to be a fairly clear consensus:

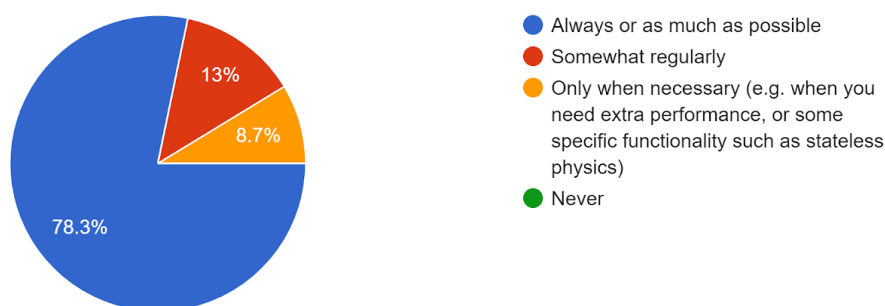
- **Jobs and Burst are usable for production** and are generally considered to not need any more improvements. They are used regularly by 100% of the respondents.



- **Entities is considered to be usable by almost everyone**, but they also are in agreement that it needs more improvements. Only one out of 23 respondents does not use Entities regularly.
- **Physics and Hybrid Renderer are considered usable by some**, but there are still some who either have no opinion or consider it unusable. Around half of respondents use these packages regularly.
- **Most people consider Animation to be unusable**, or they have no opinion. Only 4 people (17%) report to regularly use this package, and only three consider it to be usable in production.
- **Most have no opinion about the multiplayer packages Netcode and Transport**, but those who do have an opinion are evenly split between "usable and needs improvement" and "unusable for production".
- **Nobody has an opinion on DSPGraph** except for three people, and they all have different answers about its usability. Only two people report using it regularly.

How often do you use DOTS packages in your projects?

23 responses



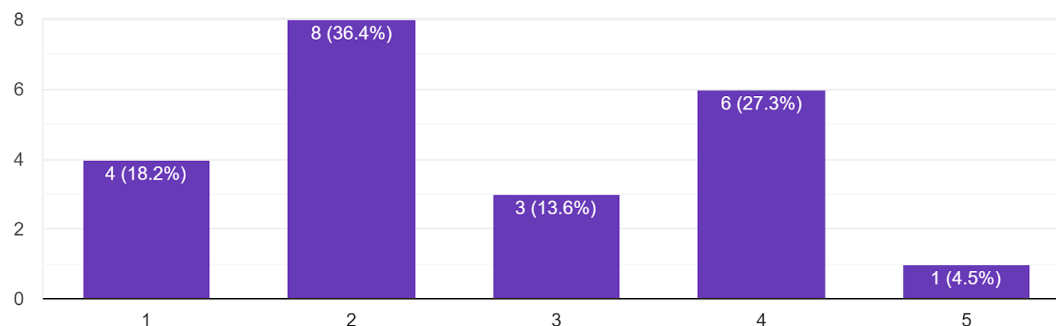
This is also very interesting: even though many packages are considered to either need more improvements or are completely unusable for production, 78% still reported to use DOTS packages as much as possible or always. 0% reported to never use DOTS packages, however this was expected as all respondents are users of the DOTS forum; someone who never uses DOTS packages obviously seems very unlikely to frequently visit this forum.



The following statements were answered with a score between 1-5, 1 meaning "disagree" and 5 meaning "agree":

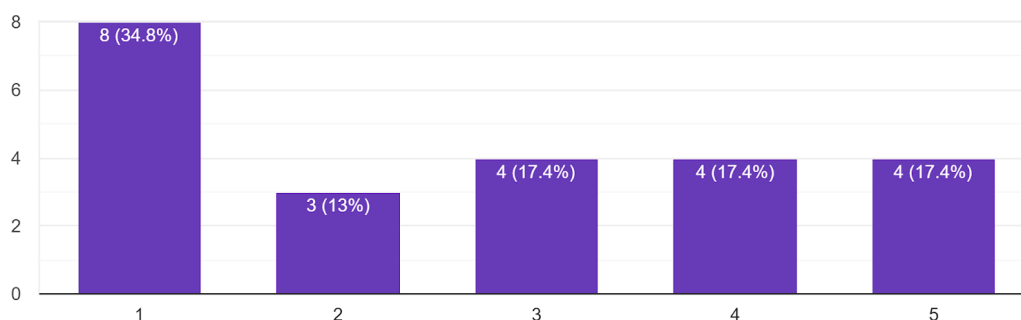
Writing code using Entities(ECS) currently takes me significantly longer than classic Unity/C#.

22 responses



Writing code that is compatible with Burst and/or Jobs currently takes me significantly longer than classic Unity or C#.

23 responses

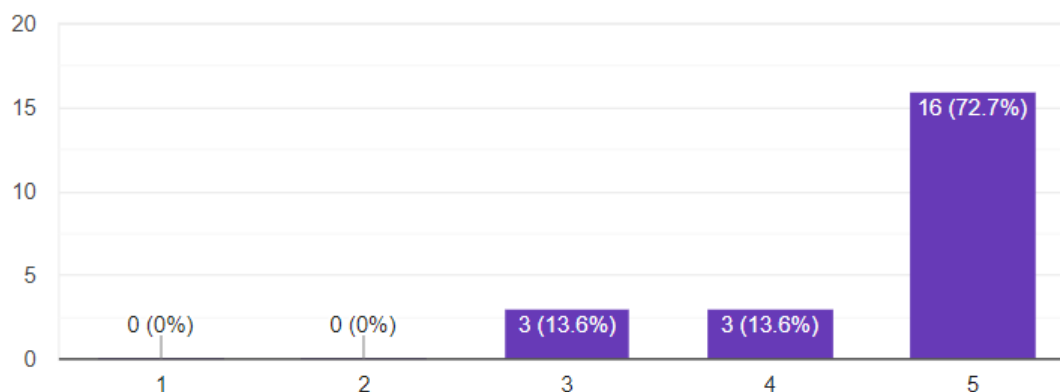


More than half(54%) of respondents does not think that writing ECS code takes longer than classic Unity/C# and 32% agree to some extent that it does take significantly longer. The results for Burst and Jobs are somewhat similar, with a total of 48% disagreeing (a score of 1-2) and 35% agreeing (a score of 4-5). The distribution is slightly different though; the opinions about Entities seem to be a bit more neutral than about Burst/Jobs: 52% chose either a 1 or 5 when answering for Burst/Jobs, as opposed to 23% for Entities.

In hindsight, these questions should have left open the possibility for writing DOTS code to be faster than classic Unity/C#. It would have been interesting to see how many people think writing DOTS code is faster. However, we do know about their preference regarding ECS/DOD versus Classic Unity/OOP:

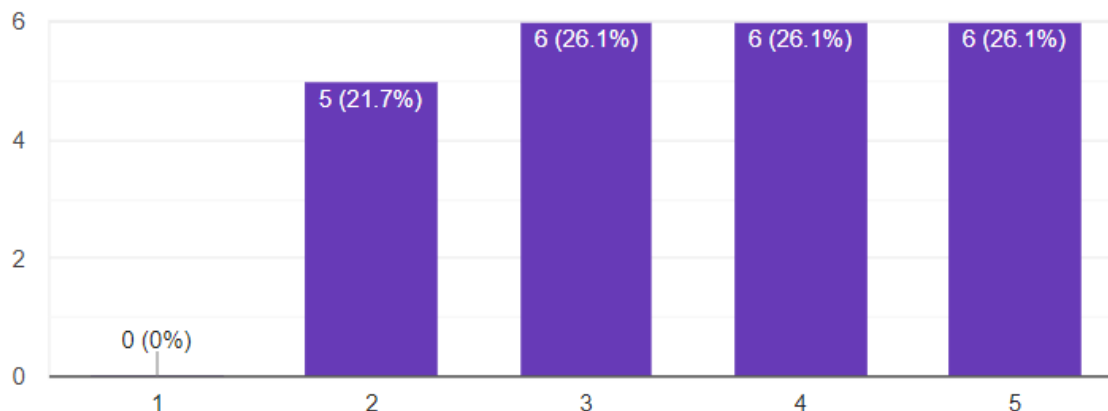
I generally prefer ECS or Data Oriented Design over Object Oriented Programming.

22 responses



I believe that one day DOTS packages, such as Entities and Unity Physics, will become the default or most used method of development in Unity.

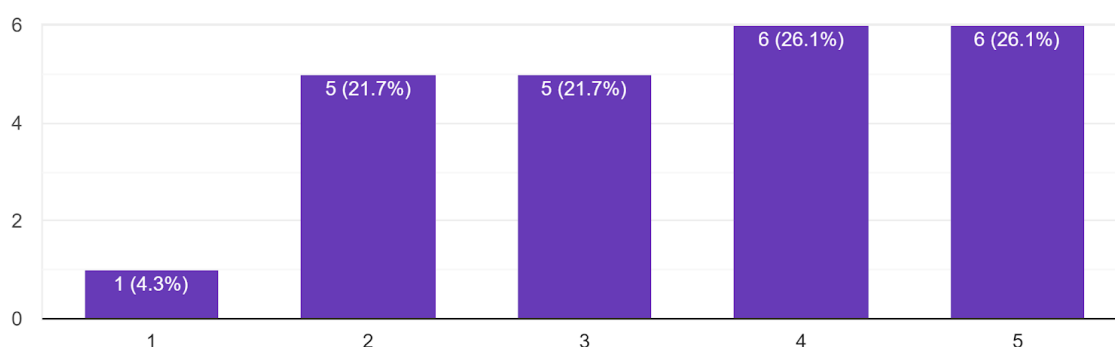
23 responses



This, once again, seems very clear: though there is still some skepticism about DOTS being the future of Unity, many do believe to some extent DOTS is the future of Unity. More importantly, the respondents clearly seem to prefer ECS/DOD over OOP. This is also confirmed by the answers to the following questions:

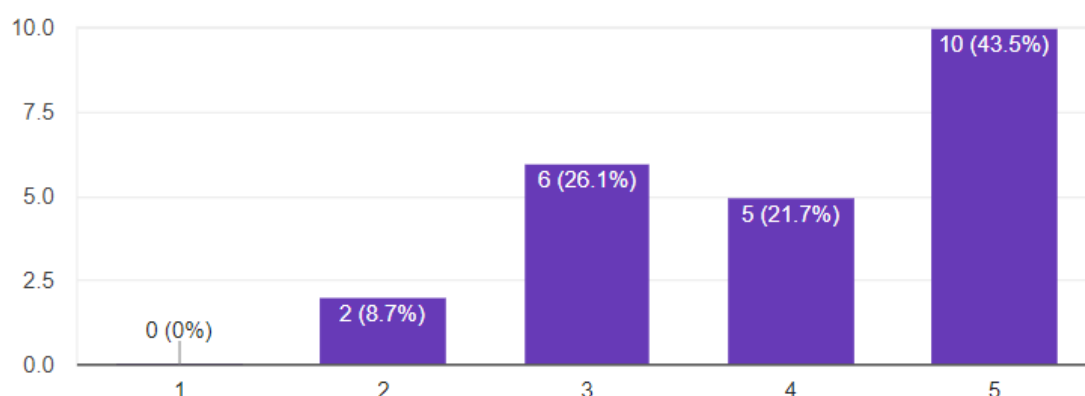
ECS code is more clear or readable to me than classic Unity code or C#.

23 responses



The reusability of ECS code more than makes up for missing OOP principles like polymorphism and inheritance.

23 responses



More than half (52%) found ECS to be more readable than classic Unity, though a quarter of respondents (26%) still seem to disagree to some extent. Only 9% found that the reusability of code is not enough to fill the gaps of some OOP functionality. In conclusion, the respondents clearly prefer ECS/DOD over OOP, and they seem to agree with the advantages of ECS mentioned by Unity Technologies and others such as the readability and reusability of code.

Final notes

At the end of the survey, there were three open questions. To prevent this chapter from getting too long, not all answers to these questions are included and some answers have been partially omitted. Otherwise, the answers have been unedited. [The link to the original survey](#) has the full answers and previously mentioned statistics, and should be accessible publicly.

Do you have any advice or recommendations for Unity developers who want to get started with DOTS?

One recurring theme was the learning curve, and the tendency for new DOTS programmers to try to use OOP where it does not belong:

"Don't give up. The learning curve is exponential."

"Stop trying to fit OOP concepts onto DOD and you'll have a much better time."

"Try to learn it 'from scratch', without any preconceived notions. Try to avoid leaning on old OO habits, this can easily make you feel like you're fighting DOTS, and you probably won't get the speed or reusability elements."

"It can be daunting for people unfamiliar with the concept, but it teaches you a lot about programming in general and pushes you outside of your comfort zone."

"You will learn along the way. Be prepared to have to revamp old systems that you wrote early on. Don't over-use DOTS when it's not needed, certain aspects of game dev are easier or more suitable in traditional coding. Learning the C# concepts about reference types, value types and garbage generation as well as regular coding concepts such as stack/heap memory, cache lines and cache hits/misses is more or less mandatory, and will help out a lot in understanding the packages and when thinking about how to implement things. When you get the hang of how to work in ECS, things are lovely, decoupled and fast to add on features to."

Many others recommend using Jobs and Burst first, and seem more reluctant to advise using "Preview" or "Experimental" packages like Entities or Physics.

"Right now, only do it if you really just want to use DOTS. If you need to make a product with a deadline, I would seriously consider the classic workflow. Burst and Jobs are ready, those can, and should, be added to the classic workflow. ECS is workable, but you are going to spend a lot of time learning undocumented code that is still missing some core features."

"Start by making Burst/Jobs, learn ecs only after familiar with previous. Because Burst/Jobs can be used in any project while ecs is project dependent."

"Read up about ECS & Jobs first."

"Start with Jobs and Burst and do something with proc-gen or some other computationally expensive task. Also, familiarize yourself with the differences between references, values, and pointers. Learning modern C (98 or newer) will help a lot."

"Jobs and Burst are for every project. Give them a try. ECS is pretty cool but you might want to give it some time to mature (animation, audio, networking, particles, etc need time)"

"First build strong foothold and understanding in using Unity and general C# programming. Only then, move to DOTS. Can start from initially implementing jobs, before moving into ECS and burst."

Have you (and/or your team) published any projects that are made using DOTS? If possible, please include what parts of the project are made using DOTS, or which specific DOTS packages were significant for development.

Only two replies mention already having published a project made with DOTS:

"Yes. Pathfinding, rendering, and some game logic."

"Most of my stuff on itch.io is made with DOTS. But the real high-quality projects haven't seen the public internet yet."

Some teams are currently developing games with DOTS that are soon to be published:

"Our team works on RTS using DOTS with modding features. We use all DOTS packages, but we are developing and improving our custom networking, rendering and animation solution, as default Unity solutions are underperforming for multi-thousands units RTS games (<https://www.sanctuary-rts.com/>)."

"Not yet published, Operation Valor is currently in it's testing phase and getting ready for Early Access. Rendering, ui, particle effects and sounds are done through Game Objects the rest is built on top of Entities."

"We're launching a large project in August. The server is pure DOTS with our own network stack as Netcode didn't really exist when we started. The client is a hybrid mix (mostly due to legacy reasons and the age of the project). Our next project will likely be a pure DOTS project."

"Not yet, but our large project at ~18 months in the making is almost pure DOTS. We have long strived to learn the foundation of the DOTS packages, extending them and experimenting with them as we go. Our world creation editor tools are even made using DOTS and the editor world, which enabled us to do things that were impossible performance wise had we used GameObjects. A nice benefit of that is that we can simulate large parts of the regular game features in the editor world too, just [by] putting some extra systems into the editor world."

Any other notes about your experience with/your opinion about DOTS?

Generally, people seem to be fairly positive about DOTS:

"I love using DOTS in procedural generation."

"I love it and it's very useful for me as we make complex games with lots of entities."

Though, some developers want more communication or more frequent updates from Unity:

"I want more information about DOTS progress. I don't want promises, just what's the current state and how is development going. Now it's been silence for 6 months and I don't like that."

"I used to be more keen about DOTS and still use it, but more begrudgingly. I'm certainly not convinced it's the future of Unity and the lack of progress recently is arguable starting to confirm this gradually."

What's interesting is that, despite a lot of packages being considered by a large part of the respondents to either be unusable for production or needing improvement, the opinions about DOTS are generally very positive. For example, the same developer who mentioned not being convinced that DOTS is the future of Unity also expressed some frustration with the limitations of the Burst compiler and stability of debugging tools, yet overall they are still positive about DOTS:

"The limitations of Burst compatible C# are very annoying and lead to me reaching for unsafe C# and pointers to work around some things, eg. a Delaunay triangulation library where I have no desire or time to port the entire thing to ECS/Jobs. It would be easier to do some things in C/C++ to be honest. Debugging Burst/Jobs is kind of a nightmare. I don't even try connecting a debugger anymore because I got fed up with things hanging/crashing. This might not be Unity's fault entirely but that's not the point. [...] It's good to be able to run fast code now with DOTS, I was severely limited previously by the old Unity C# speed. This is a massive bonus. I can now run code as fast as I did a few years ago when I made games in C/C++"

It seems that most respondents are very positive about DOTS and though many are waiting for improvements or specific functionalities, most are still glad to be able to use it in its current state.

"As long as the ECS team can get some big pain points taken care of (such as heavy main thread costs of `SystemBase.ShouldRunSystem` and expensive scheduling of jobs), the packages will be great. I don't mind the current API too much as I've already learned it, but I can see that beginners might have a harder time wrapping their head around it."

Attachment 2: Prototype

A prototype has been written to ensure that first of all, the advice given is not just from theoretical research and second-hand opinions, but also from first-hand experience. Writing the prototype also provided some guidance to prioritizing the overwhelming amount of information found in Unity's manuals and documentation. It also showed some parts that I felt were missing from the manual, such as referencing entities. Furthermore, the problems I ran into can be very major problems for development: being able to document these and possibly prevent these in the future was definitely beneficial.

The prototype is fairly simple: it is a 3d platformer where the player can move using the WASD keys, avoid obstacles and collect items. Some functionality, such as checking if an entity is touching the ground or linking an entity to a game object, have several implementations showing different solutions to these problems.

The prototype, and this advisory report, can be found on GitHub:

https://github.com/Rebels-io/DOTS_Prototype



Attachment 3: Benchmarks

The following is a collection of measurements done by various members of the Unity community, to get an indication for what kind of performance increases are possible when using DOTS.

The clearest example you can find to see the increase in performance per package [Far North's example](#) from Unite Copenhagen 2019.

Stage	Count	Time	Perf. Factor
Before DOTS Implementation	2 000	9 ms	1x
ECS	2 000	1 ms	9x
ECS + Job System	2 000	0.2 ms	35x
ECS + Jobs + Burst Compiler	20 000	0.04 ms	2250x

Another example that Unity Technologies has showcased is [BigBox VR's Population One](#):

'Population: One' is a VR multiplayer First-Person Shooter designed from the ground up for vertical combat. 18 players on a one kilometer map, running at 72 frames per second on the Oculus Quest and PC VR headsets. 'Population: One' is already a big project for PC VR, but without DOTS, Addressables and the Scriptable Render Pipeline, Population: One simply wouldn't have been possible on the Quest. We used the DOTS framework heavily to achieve multi-core utilization to achieve a steady 72 frames per second in Oculus Quest and with our custom physics and LOD engine built with Burst and the C# Job System, we achieved this system that's up to 20 times faster than the original implementation.

Keep in mind that as these were showcased by Unity Technologies, some confirmation bias is expected in these previous examples. However, they don't seem to be inaccurate when compared to other sources.

Code Monkey's tutorial "[Game Objects with Unity DOTS Pathfinding!](#)" shows several GameObjects using pathfinding simultaneously with a Hybrid DOTS implementation in only 0.016 milliseconds.



[Unity DOTS vs MonoBehaviour performance test](#) by RGEL shows a school of fish being simulated. With a fish count of 3000, his GameObject implementation runs a simulation of fish with about 45 FPS average. The DOTS equivalent simulates the same amount of fish at 70 FPS. Doubling the amount of fish has a massive impact on GameObject performance, dropping to 25 FPS; yet the DOTS version still stays at exactly 70 FPS average. Doubling yet again to a total of 12000 fish makes the FPS of the GameObject implementation severely drop once more, this time to an average of 11 FPS. Again, no impact on the DOTS variant. Increasing the amount of fish to a total of 35000 finally starts to impact the performance of the DOTS variant, dropping the total amount of frames to around 27 FPS, while the GameObject implementation is struggling at a mere 4 FPS. At around 50 000 fish, the GameObject implementation completely crashes, yet the DOTS variant is still running at 18 FPS. After adjusting some lighting settings, he is once more able to run at a smooth (enough) 30 FPS, even though there are 120.000 visible fish in the scene.

The [DOTS prototype](#) made for Rebels contains a scene for testing the performance of moving objects using sine waves (*Scenes > PerformanceMeasurements > SineWaveCubes.unity*). Open the Profiler window (CTRL + 7) and choose "Hierarchy" in the bottom half. Use the search bar to look for "Sine", and you will see the performance of both the MonoBehaviour Update and the ECS system update. These are the measurements on my computer:

Amount	Total time (Game Objects)	Total time (Entities)	Performance increase
1	0.00 ms	0.03 ms	n/a (+0.03 ms)
10	0.01 ms	0.03 ms	0.3x (+0.02 ms)
100	0.06 ms	0.03 ms	2x (-0.03 ms)
1 000	0.50 ms	0.03 ms	7x (-0.47 ms)
5 000	2.94 ms	0.17 ms	17x (-0.03 ms)
10 000	5.72 ms	0.24 ms	24x (-5.48 ms)