→ E-BOOK

Introduction to DOTS concepts, features, & samples for advanced Unity developers

(Unity 6 edition)

Unity®

Contents

Introduction **5**

Author and expert contributors 5

About performance **6**

DOTS packages and features **9**

The C# job system 9

Scheduling and completing jobs 11

Job safety checks and dependencies 12

The Burst compiler 13

Collections 15

Mathematics 15

Entities (ECS) 16

Archetypes 16

Chunks 17

Queries 17

Job system integration 18

Subscenes and baking 19

Streaming 21

Entities Graphics 22

Physics 22

Netcode for Entities 23

Authoritative server 23

Client-side prediction 23

Character Controller 25

Animation 26

User Interfaces 26

DOTS educational content 27

Evaluating DOTS for your project 30

For existing projects 30

For new projects 31

Made with DOTS 32

Made with DOTS: *Bare Butt Boxing*, by Tuatara Games. . 33

Made with DOTS: *Histera*, by StickyLock Games. 34

Made with DOTS: *V Rising*, by Stunlock Studios 35

Made with DOTS: *Zenith: The Last City*, by Ramen VR . . 36

Made with DOTS: *Den of Wolves*, by 10 Chambers 37

Made with DOTS: Megacity Metro sample 39

Appendix I: Misconceptions about DOTS and Unity Entities. 40

False: DOTS, ECS, Unity Entities, and data-oriented design are all the same thing. 40

False: Using DOTS requires using entities 41

False: Using DOTS will make any Unity game significantly faster. 41

False: Every new Unity project should use entities 42

False: The benefits of ECS are just about performance . 42

False: The performance benefit of entities/ECS is all about memory efficiency and cache utilization. . . . 42

False: Entities are the ultimate, optimal data structure for everything 43

False: Multithreaded programming is too hard for most programmers. 44

False: Manual memory management is too hard for most programmers. 45

False: GameObjects and Entities cannot be used together, and (Mostly) False: Entities cannot animate, emit sound, or do UI, etc. 45

(Mostly) False: DOTS code cannot use managed objects. 46

Appendix II: Hardware concepts related to performance . . . 47

Memory allocation and garbage collection. 47

Multithreaded programming 49

Memory and CPU cache. 50

Appendix III: Writing software for performance 52

Costs of object-oriented programming. 52

Performance costs of OOP 53

Structural costs of OOP. 54

Data-oriented design 55

Design your data before designing your code 56

Prefer simple data 56

Think of your code as a data pipeline 56

Measure, estimate, and budget performance at all stages of development. 57

Prefer specific solutions over abstractions 58

More advanced resources from Unity. 59

Introduction

This guide explains the potential performance benefits of Unity's Data-Oriented Technology Stack (DOTS). It provides a high-level overview of each of the packages and features included in the stack, as well as explaining some of the core concepts and areas related to, and impacted by, design-oriented design (DOD). It doesn't go into the details of the API's, but you will find links throughout to many new DOTS tutorials and other learning resources where you can learn more.

Our primary goal with the e-book is to provide you with the knowledge you need to make an informed decision about whether your Unity project will benefit from using some or all of the DOTS features. Secondly, we aim to make it easier for you to dive into our samples and other educational resources and get started using DOTS.

Author and expert contributors

This e-book was created from a collaboration between Unity DOTS engineers and external experts. The main author is Brian Will, a software engineer at Unity. Other experts who contributed to this guide are:

— Daniel Kierkegaard Andersen, software engineer, Unity

— Laurent Gibert, director, product management, Unity

— Thomas Krogh-Jacobsen, Technical Content Marketing

— Nik Lever, real-time 3D and Unity educator

— Steve McGreal, software engineer

About performance

As experienced game developers know, performance optimization is not an optional part of game development. Maybe your game performs nicely on a high-end PC, but what about the low-end mobile platforms you're also targeting? Do some frames take much longer than others and thus create noticeable hitches? Are loading times annoyingly long? Does the game freeze for full seconds every time the player walks through a door? Such performance problems not only detract from the player experience, they may effectively prohibit you from adding more features: more environment detail and scale, more mechanics, more characters and character behaviors, more physically-simulated objects, and possibly even more release platforms.

What's the culprit? In many projects, it's rendering: maybe the textures are too large, the meshes too complex, the shaders too expensive, or maybe the rendering makes ineffective use of batching, culling, or LOD. Another common pitfall is excessive use of complex mesh colliders, which greatly increase the cost of the physics simulation.

In other cases, though, the game simulation itself is slow: The C# code you wrote that defines what makes your game unique is taking too many milliseconds of CPU time per frame. The only fix, then, is to make your game code fast…or at least not slow. But how?

In previous decades, PC game developers could often solve this problem by just waiting for new technology to become available. From the 1970s and into the 21st century, CPU single-threaded performance generally doubled every few years, so a PC game would "magically" get faster over its life cycle. In the last two decades, however, CPU single-threaded performance gains have been relatively modest as we get closer to physical limitations. Moreover, the gap between high-end and low-end gaming devices has widened, with a large chunk of the player base using hardware that is several years old. Waiting for faster hardware no longer seems like a workable strategy.

The question to ask, then, is “Why is my CPU code slow in the first place?” There are several common issues:

— **Garbage collection induces noticeable overhead and pauses:** This occurs because the garbage collector serves as an automatic memory manager that manages the allocation and release of memory for an application. Not only does garbage collection incur CPU and memory overhead, it sometimes pauses all execution of your code for many milliseconds. Users might experience these pauses as small hitches or more intrusive stutters.

— **The compiler-generated machine code is suboptimal:** Some compilers generate much less optimized code than others, with results varying across platforms.

— **The CPU cores are insufficiently utilized:** Although today’s lowest-end devices have multi-core CPUs, many games simply keep most of their logic on the main thread because writing multithreaded code is often difficult and prone to error.

— **The data is not cache friendly:** Accessing data from cache is much faster than fetching it from main memory. However, accessing system memory may require the CPU to sit and wait for hundreds of CPU cycles; instead, you want the CPU to read and write data from its cache as much as possible.

  The simplest way to arrange this is to read and write memory sequentially, and so the most cache-friendly way to store data is in tightly-packed, contiguous arrays. Conversely, if your data is strewn non-contiguously throughout memory, accessing it will typically trigger many expensive cache misses; the CPU requests data that is not present in the cache memory and instead needs to fetch it from the slower main memory

— **The code is not cache friendly:** When code is executed, it must be loaded from system memory if it’s not already sitting in cache. One strategy is to favor calling a function in as few places as possible to reduce how often it must be loaded from system memory. For example, rather than call a particular function at various places strewn throughout your frame, it’s better to call it in a single loop so that the code only needs to be loaded at most once per frame.

— **The code is excessively abstracted:** Among other issues, abstraction tends to create complexity in both data and code, which exacerbates the aforementioned problems: managing allocations without garbage collection becomes harder; the compiler may not be able to optimize as effectively; safe and efficient multithreading becomes harder, and your data and code tend to become less cache-friendly. On top of all this, abstractions tend to spread around performance costs, such that the whole code is slower, leaving you with no clear bottlenecks to optimize.

All of the above ailments are commonly found in Unity projects, for several reasons:

— Although C# allows you to create manually-allocated objects (meaning objects which are not garbage collected), the default norm in C# and most Unity projects is to use **C# class instances**, which are garbage collected. In practice, Unity users have long mitigated this issue with a technique called pooling (even though pooling arguably

defeats the purpose of using a garbage-collected language in the first place). The main benefit of object pooling is the efficient reuse of objects from a preallocated pool, eliminating the need for frequent creation and deallocation of objects.

- In the Unity Editor, C# code is normally compiled to machine code with the **Mono Compiler.** For standalone builds you can generally get better results using IL2CPP (C# Intermediate Language cross-compiled to C++), but this brings some downsides, like longer build times and making mod support more difficult.
- It's common that Unity projects **run all their code on the main thread**, partly because doing so is what Unity makes easy:
  - The Unity event functions, such as the Update() method of MonoBehaviours, are all run on the main thread.
  - Most Unity APIs can only be safely called from the main thread.
- The data in a typical Unity project tends to be structured as **a bunch of random objects scattered throughout memory**, leading to very poor cache utilization. Again, this is partly because this is what Unity makes easy:
  - A GameObject and its components are all separately allocated, so they often end up in different parts of memory.
- The code in a typical Unity project tends to **not be cache friendly**:
  - Conventional C# and Unity's APIs encourage an object-oriented style of code, which tends towards numerous small methods and complex call chains. Unlike a data-oriented approach it's not very hardware friendly.
  - The event functions of every MonoBehaviour are invoked individually, and the calls are not necessarily grouped by MonoBehaviour type. For example, if you have 1000 Monster MonoBehaviours, each Monster is updated separately and not necessarily along with the other Monsters.
- The object-oriented style of conventional C# and many Unity APIs generally lead to **abstraction-heavy solutions**. The resulting code then tends to have inefficiencies laced throughout that are hard to disentangle and isolate.

For more background information on these issues, see the appendix at the end of this guide that covers the following concepts:

- Memory allocation and garbage collection
- Multithreaded programming
- Memory and CPU cache
- Object-oriented programming and abstraction
- Data-oriented design

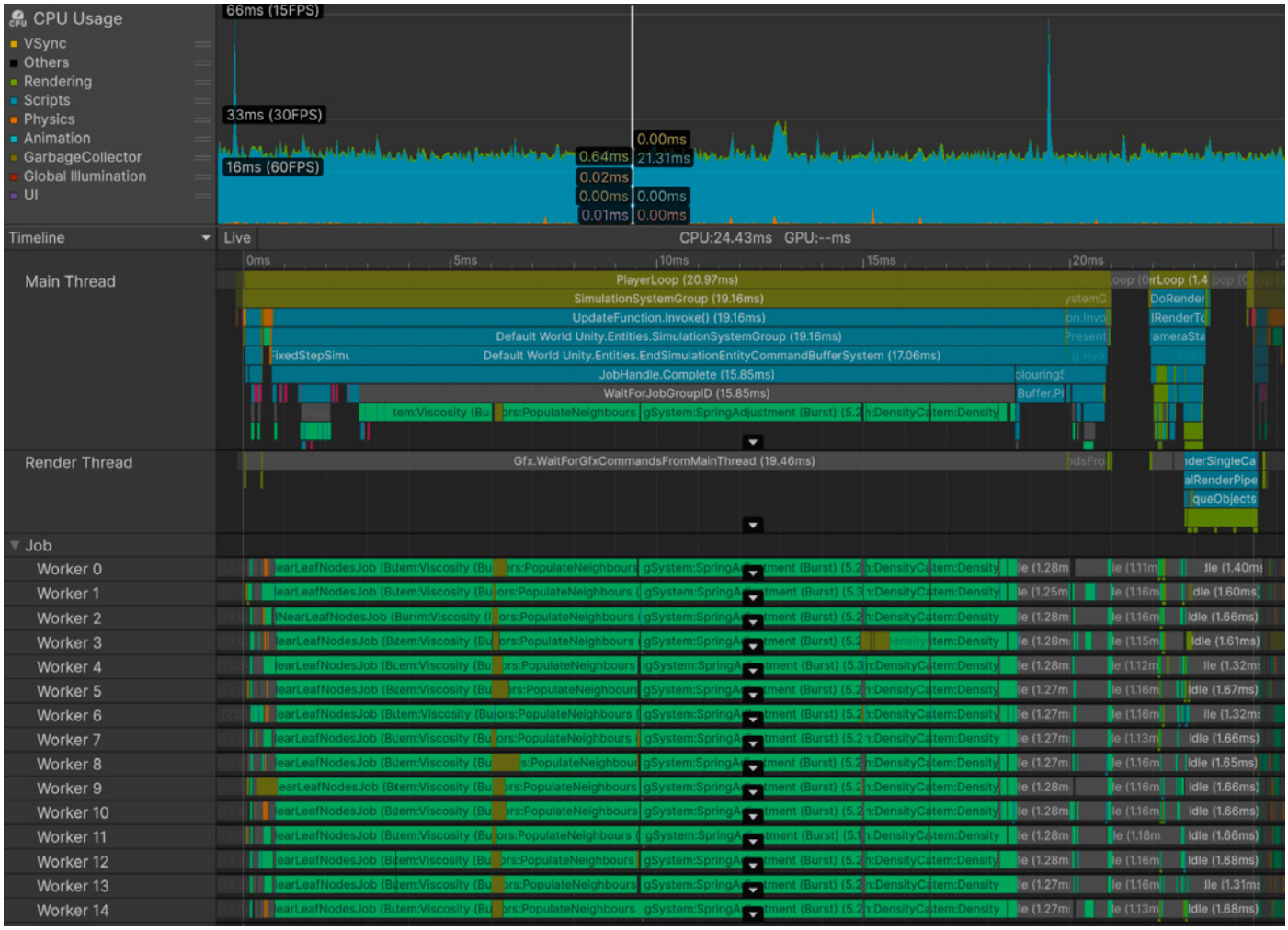
DOTS packages and features

This section outlines the DOTS packages and features.

The C# job system

The C# job system provides an easy and efficient way to write multithreaded code that helps your application take advantage of all available CPU cores.

Unlike the other features of DOTS, the job system is not a package but included in the Unity core module.



A profile showing Burst-compiled jobs utilizing the CPU and running across many worker threads

Because MonoBehaviour updates are executed only on the main thread, many Unity games end up running all of their game logic on just one CPU core. To take advantage of additional cores, you could manually spawn and manage additional threads, but doing so safely and efficiently can be very difficult.

For an easier alternative, Unity provides the C# job system:

— The job system maintains a pool of worker threads, one for each additional core of the target platform. For example, when Unity runs on eight cores, it creates one main thread and seven worker threads.

— The worker threads execute units of work called jobs. When a worker thread is idle, it pulls the next available job from the job queue to execute.

— Once a job starts execution on a worker thread, it runs to completion. (In other words, jobs are not preempted.)

```
// A simple example job that multiplies the
// elements of two arrays.
// Implementing IJob makes this struct a job type.
struct MyJob : IJob
{
    // A NativeArray is “unmanaged”, meaning it
    // isn’t garbage collected.
    public NativeArray<float> Input;
    public NativeArray<float> Output;

    // The Execute method is called when the
    // job system executes this job.
    public void Execute()
    {
        // Multiply every value in Output by the
        // corresponding value in the Input array.
        for (int i = 0; i < Input.Length; i++)
        {
            Output[i] *= Input[i];
        }
    }
}
```

Scheduling and completing jobs

— Jobs can only be scheduled (meaning, added to the job queue) from the main thread, not from other jobs.

— When the main thread calls the `Complete()` method on a scheduled job, it waits for the job to finish execution (if it hasn’t finished already).

— Only the main thread can call `Complete()`.

— After `Complete()` returns, you can be sure that the data used by the job is once again safe to access on the main thread and safe to be passed into subsequently scheduled jobs.

Job safety checks and dependencies

In multithreaded programming, ensuring safety and managing dependencies between threads are critical for avoiding race conditions, data corruption, and other concurrency issues. It's beyond the scope of this guide to explain these pitfalls. The key takeaway is to understand how the job system handles safety checks and dependencies:

- For guaranteed isolation, each job has its own private data that the main thread and other jobs can't access.
- However, jobs may also need to share data with each other or the main thread. Jobs that share the same data should not execute concurrently because this creates race conditions. So the job system "safety checks" throw errors when you schedule jobs that might conflict with others.
- When scheduling a job, you can declare that it depends upon prior scheduled jobs. The worker threads will not start executing a job until all of its dependencies have finished execution, allowing you to safely schedule jobs that would otherwise conflict.
    - For example, if jobs A and B both access the same array, you could make job B depend upon job A. This ensures job B will not execute until job A has finished, thus avoiding any possible conflict.
- Completing a job also completes all of the jobs it depends upon, directly and indirectly.

Many Unity features internally use the job system, so you will see more than just your own scheduled jobs running on the worker threads in the Profiler.

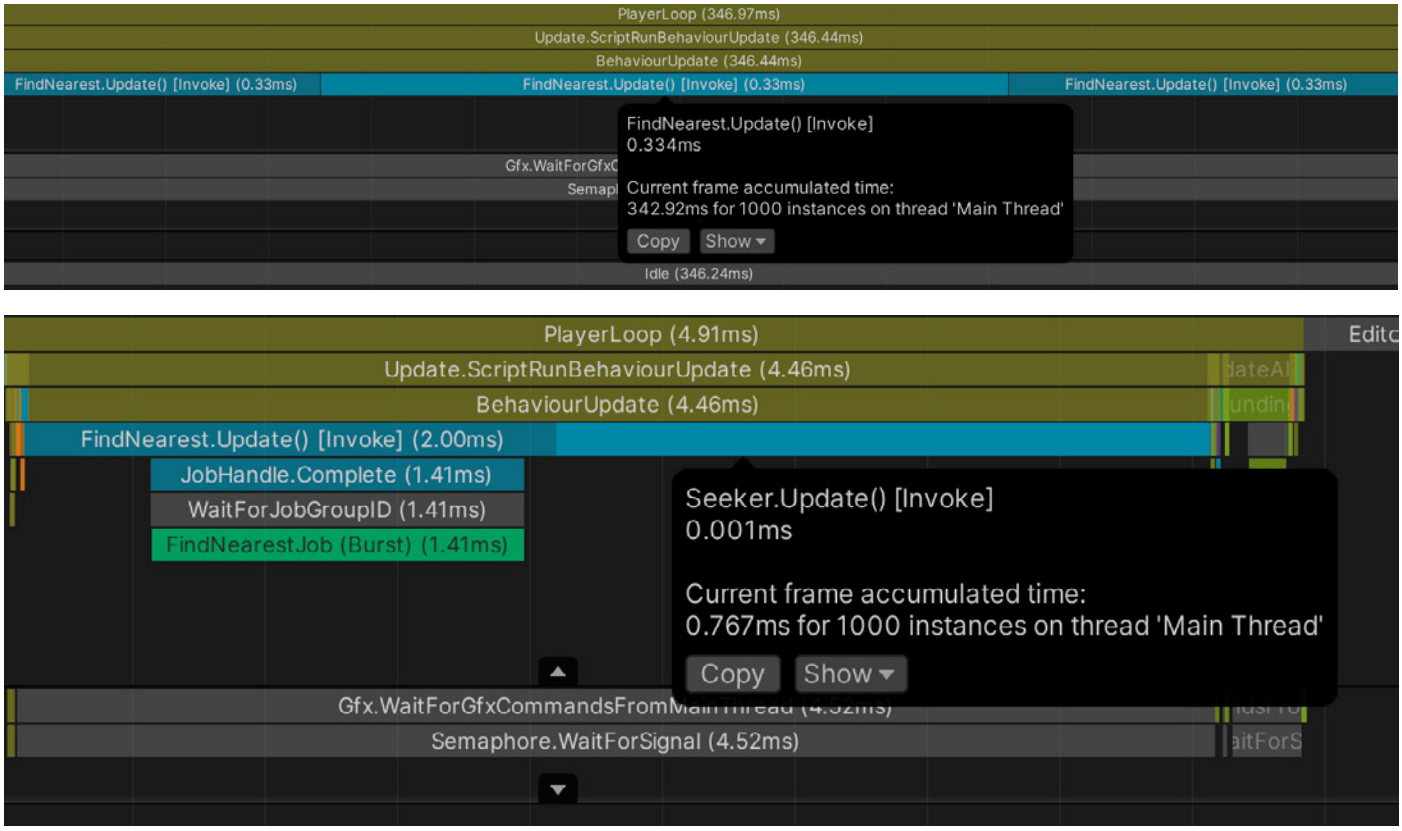
Note that jobs are intended only for processing data in memory, not performing I/O (input and output) operations, such as reading and writing files or sending and receiving data over a network connection. Because some I/O operations may block the calling thread, performing them in a job would defeat the goal of trying to maximize utilization of the CPU cores. If you want to do multithreaded I/O work, you should call asynchronous APIs from the main thread or use conventional C# multithreading.

To learn about jobs, see the Job System 101 section in the samples repo (there is also a version of the job tutorial on Unity Learn).

The Burst compiler

As stated earlier, C# code in Unity is by default compiled with Mono, a JIT (just-in-time) compiler or, alternatively with IL2CPP, an AOT (ahead of time) compiler which generally gives better runtime performance and may be better supported on some target platforms.

The Burst package provides a third compiler that performs substantial optimizations, often yielding dramatically better performance than Mono or even IL2CPP. Using Burst can greatly improve the performance and scalability of a heavy computation problem, as the following images illustrate:



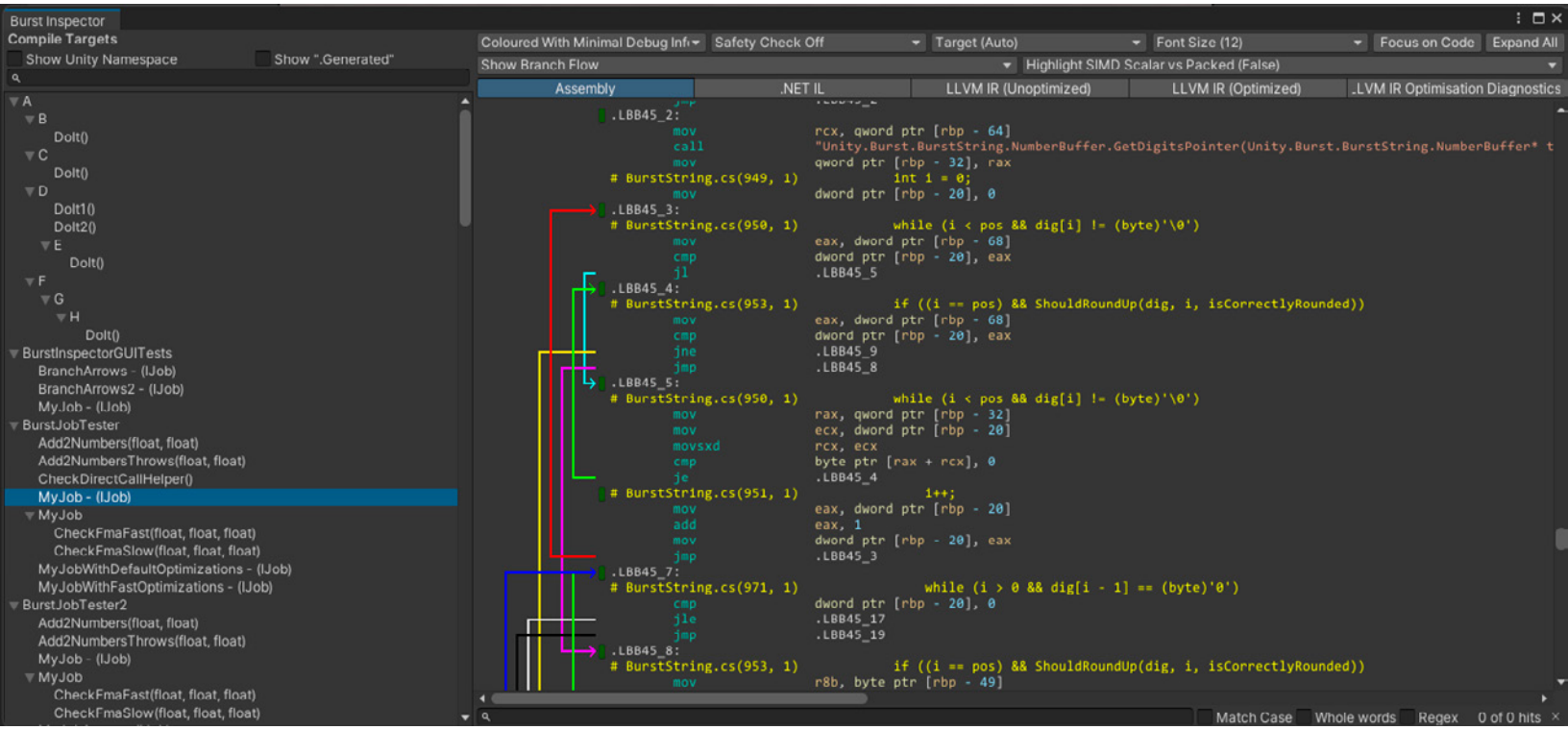
Top image: From the jobs tutorial, the FindNearest updates, compiled with Mono, take 342.9 ms. Bottom image: From the same jobs tutorial, the FindNearestJob, compiled with Burst, takes 1.4 ms.

Understand, however, that Burst can only compile a subset of C#, so a lot of typical C# code can't be compiled with it. The main limitation is that Burst-compiled code can't access managed objects, including all class instances. As this excludes most conventional C# code, Burst compilation is only applied selectively to designated parts of code, such as jobs:

```
// A Burst-compiled version of the previous example job.
// The BurstCompile attribute marks this job to be Burst-compiled.
[BurstCompile]
struct MyJob : IJob
{
```

```
    public NativeArray<float> Input;
    public NativeArray<float> Output;
    public void Execute()
    {
        for (int i = 0; i < Input.Length; i++)
        {
            Output[i] *= Input[i];
        }
    }
}
```

As described in this video, the performance gains of Burst come from the use of SIMD (a technique used to perform the same operation on multiple data elements simultaneously) and better awareness of aliasing (when two or more pointers or references refer to the same memory location), among other techniques.



For expert users, Burst provides a few advanced features, such as intrinsics and the Burst Inspector (pictured above), which shows the generated assembly code.

Collections

The Collections package provides unmanaged collection types, such as lists and hash maps which are optimized for usage in jobs and Burst-compiled code.

By “unmanaged”, it’s meant that these collections are not managed by the C# runtime or garbage collector; you are responsible for explicitly deallocating any unmanaged collection that you create by calling its `Dispose()` method once it’s no longer needed.

Because these collections are unmanaged, they don’t create garbage collection pressure, and can be safely used in jobs and Burst-compiled code.

The collection types fall into a few categories:

— The types whose names start with **Native** will perform safety checks. These safety checks will throw an error if the collection is not properly disposed of and/or if the collection is used with jobs in a way that isn’t thread-safe.

— The types whose names start with **Unsafe** perform no safety checks.

— The remaining types which are neither **Native** or **Unsafe** are small struct types with no pointers, so they are not allocated at all. Consequently, they need no disposal and have no potential thread-safety issues.

Several **Native** types have **Unsafe** equivalents. For example, there is both **NativeList** and **UnsafeList**, and both **NativeHashMap** and **UnsafeHashMap**, among other pairs. For the sake of safety, you should prefer using the **Native** collections over the **Unsafe** equivalents when you can.

Mathematics

The Mathematics package is a C# math library that, similar to Collections, is created for Burst and the job system to be able to compile C#/IL code into highly efficient native code. It provides you with:

— Vector and matrix types, such as **float3**, **quaternion**, **float3×3**

— Many math methods and operators that follow HLSL-like shader conventions

— Special Burst compiler optimization hooks for many methods and operators

See this Unity.Mathematics cheat sheet for more information.

Note that most types and methods of the old **UnityEngine.Mathf** library are usable in Burst-compiled code, but the **Unity.Mathematics** equivalents will perform better in some cases.

Entities (ECS)

The Entities package provides an implementation of ECS, an architectural pattern composed of **entities** and **components** for data and **systems** for code.

In short, an entity is composed of components, where each component is usually a C# struct. Like with GameObjects, an entity's components can be added and removed over its lifetime.

Unlike with GameObjects, an entity's components do not usually have their own methods. Instead, in ECS, each "system" has an update method that is invoked usually once per frame, and these updates will read and modify the components of some entities. For example, a game with monsters might have a MonsterMoveSystem whose update method modifies the Transform components of every monster entity.

Archetypes

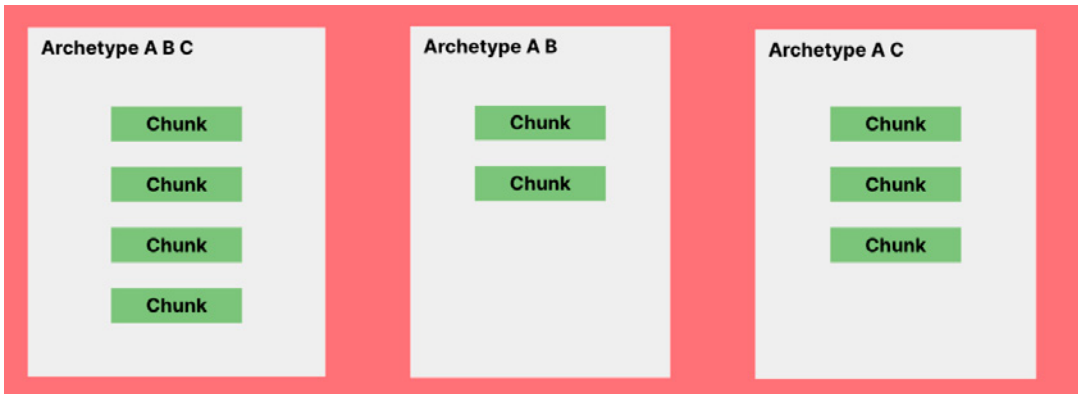
In Unity's ECS, all entities with the same set of component types are stored together in the same "archetype". For example, say you have three component types: A, B, and C. Each unique combination of component types is a separate archetype, e.g.:

— All entities with component types A, B, and C, are stored together in one archetype.

— All entities with component types A and B are stored together in a second archetype.

— All entities with component types A and C are stored in a third archetype.

Adding a component to an entity or removing a component from an entity moves the entity to a different archetype.



In Unity's ECS, all entities with the same set of component types are stored together in the same "archetype".

Chunks

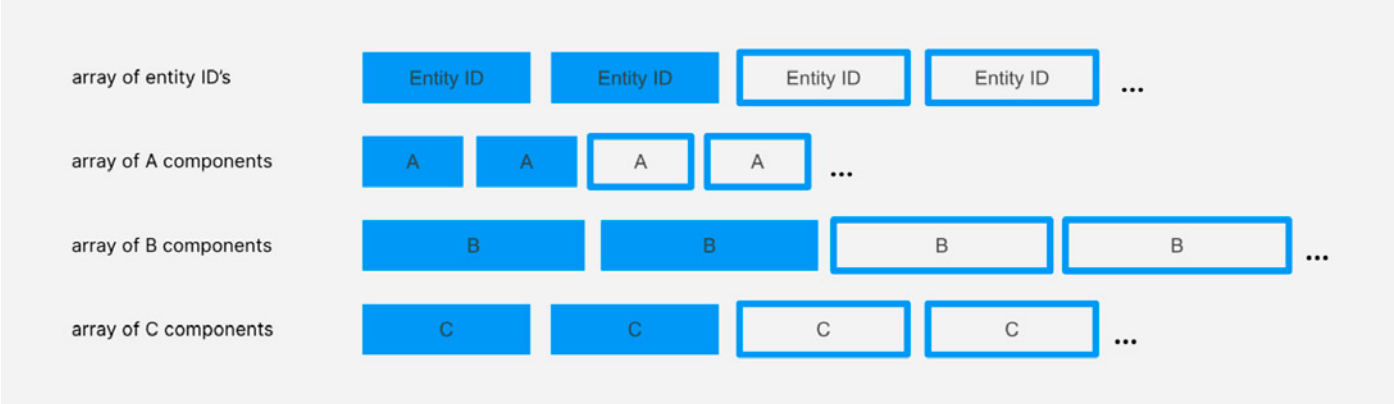
Within an archetype, the entities and their components are stored in blocks of memory called chunks. Each chunk stores up to 128 entities, and the components of each type are stored in their own array within the chunk. For example, in the archetype for entities having component types A and B, each chunk will store three arrays:

— one array for the entity ID's

— a second array for the A components

— and a third array for the B components

The ID and components of the first entity in a chunk are stored at index 0 of these arrays, the second entity at index 1, the third entity at index 2, and so on.



How chunks work in Unity's ECS architecture

A chunk's arrays are always kept tightly packed:

— When a new entity is added to the chunk, it's stored in the first free index of the arrays.

— When an entity is removed from the chunk, the last entity in the chunk is moved to fill in the gap (An entity is removed from a chunk when it's being destroyed or moved to another archetype.)

Queries

A primary benefit of the archetype- and chunk-based data layout is that it allows for efficient querying and iteration of the entities.

To loop through all entities having a certain set of component types, an entity query first finds all archetypes matching that criteria, and then it iterates through the entities in the archetypes' chunks:

— Since the components in the chunks reside in tightly packed arrays, looping through the component values largely avoids cache misses.

— Since the set of archetypes tends to remain stable throughout most of a program, the set of archetypes matching a query can usually be cached to make the queries even faster.

```
// A simple example system.
public partial struct MonsterMoveSystem : ISystem
{
    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // Query that loops through all entities with
        // a LocalTransform, Velocity, and Monster component
        foreach (var (transform, velocity) in
            SystemAPI.Query<RefRW<LocalTransform>, RefRO<Velocity>>()
                .WithAll<Monster>())
        {
            // Update the transform position from the
            // velocity (factoring in delta time)
            transform.ValueRW.Position +=
                    velocity.ValueRO.Value * SystemAPI.Time.deltaTime;
        }
    }
}
```

Job system integration

As long as entity component types are unmanaged, they can be accessed in Burst-compiled jobs. Two special job types are provided for accessing entities: IJobChunk and IJobEntity.

```
// A simple example system that schedules an IJobEntity.
public partial struct MonsterMoveSystem : ISystem
{
    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // Create and schedule the job.
        var job = new MonsterMoveJob {
            DeltaTime = SystemAPI.Time.DeltaTime
        };
        job.ScheduleParallel();
    }
}
```

```
// A Burst-compiled job that processes every entity that has
// a LocalTransform, Velocity, and Monster component.
[WithAll(typeof(Monster))]
[BurstCompile]
public partial struct MonsterMoveJob : IJobEntity
{
    public float DeltaTime;

    // Because we wish to modify the LocalTransform, we use ‘ref’.
    // We only wish to read the Velocity, so we use ‘in’.
    public void Execute(ref LocalTransform, in Velocity)
    {
        transform.Position += velocity.Value * DeltaTime;
    }
}
```

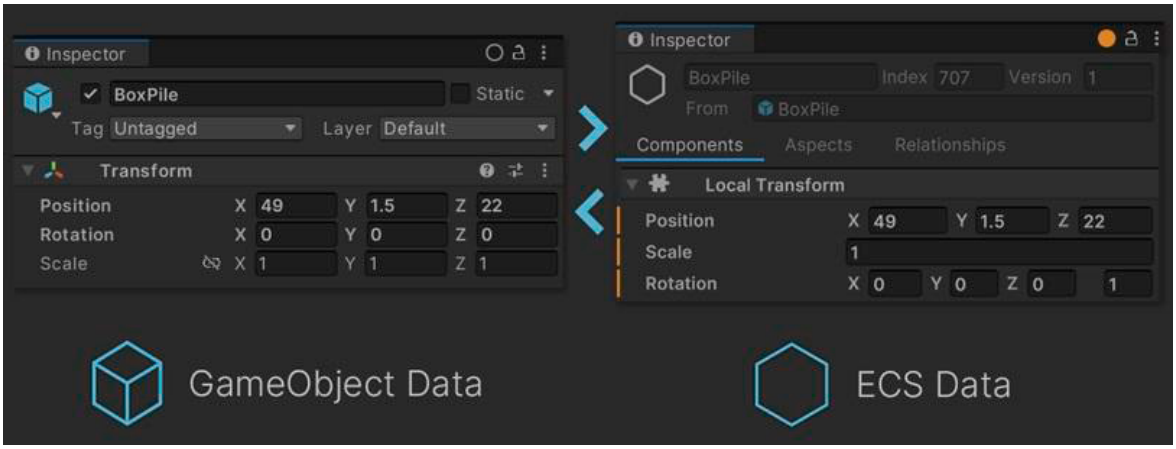
For ease of use, systems can automatically handle job dependencies and job completion across systems.

Subscenes and baking

Unity ECS uses subscenes instead of scenes to manage the content of your application. This is because Unity’s core scene system is incompatible with ECS.

While entities can’t be directly included in Unity scenes, a feature called baking allows for loading entities from scenes and converts the GameObjects and MonoBehaviour components into entities and ECS components.

You can think of subscenes as scenes that are nested inside others and are processed by baking, which re-runs every time you edit a subscene. For every GameObject in a subscene, baking creates an entity, the entities get serialized into a file, and it’s these entities that are loaded at runtime when the subscene is loaded, not the GameObjects themselves.



Left: Inspecting a GameObject and right: Inspecting an entity that was baked from the GameObject

Which components get added to the baked entities is determined by the “bakers” associated with the GameObject components. For example, bakers associated with the standard graphics components, like MeshRenderer, will add graphics-related components to the entity. For your own MonoBehaviour types, you can define bakers to control what additional components get added to the baked entities.

```
// This entity component type represents an energy shield with hit
points,
// maximum hit points, recharge delay, and recharge rate.
public struct EnergyShield : IComponentData
{
    public int HitPoints;
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;
}

// A simple example authoring component.
// An authoring component is just an ordinary MonoBehaviour
// that has a defined Baker class.
public class EnergyShieldAuthoring : MonoBehaviour
{
    public int MaxHitPoints;
    public float RechargeDelay;
    public float RechargeRate;

    // The baker for our EnergyShield authoring component.
    // This baker is run once for every EnergyShieldAuthoring
    // instance that’s attached to any GameObject in a subscene.
    class Baker : Baker<EnergyShieldAuthoring>
    {
        public override void Bake(EnergyShieldAuthoring authoring)
        {
            // The TransformUsageFlags specify which
            // transform components the entity should have.
            // The None flag means that it doesn’t need transforms.
            var entity = GetEntity(TransformUsageFlags.None);
```

```
            // This simple baker adds just one component to the entity.
            AddComponent(entity, new EnergyShield
            {
                HitPoints = authoring.MaxHitPoints,
                MaxHitPoints = authoring.MaxHitPoints,
                RechargeDelay = authoring.RechargeDelay,
                RechargeRate = authoring.RechargeRate,
            });
        }
    }
}
```

On the one hand, it's inconvenient in simple cases to not be able to add entities directly in scenes, but on the other hand, the baking process can be useful in more advanced cases. Baking effectively separates authoring data (the GameObjects that you edit in the Editor) from runtime data (the baked entities), so what you directly edit and what gets loaded at runtime don't have to match 1-to-1. For example, you could write code to procedurally generate data during baking, which would spare you from paying the cost at runtime.

Streaming

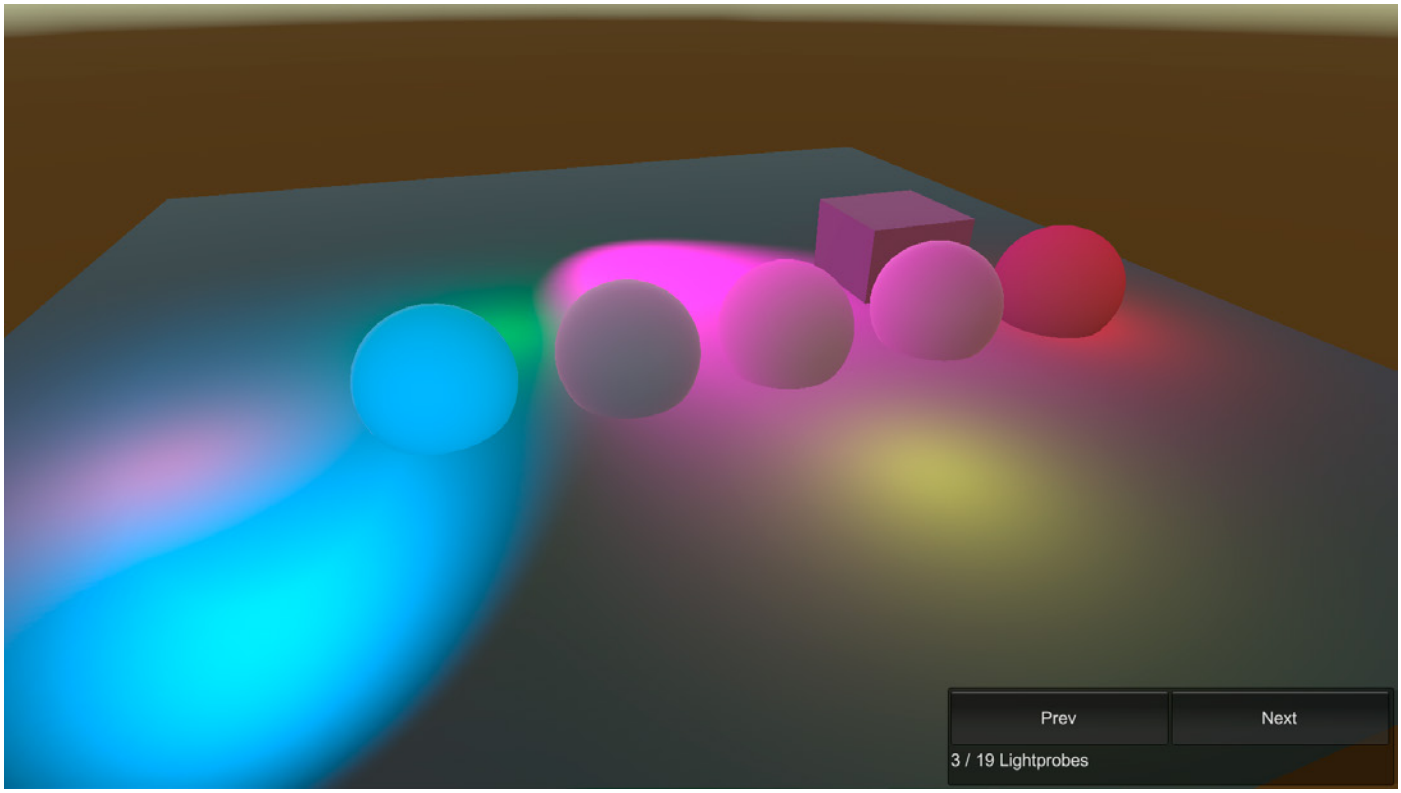
Particularly for large detailed environments, it's important to be able to load and unload many elements efficiently and asynchronously as the player or camera moves around the environment. In a large open world, for example, many elements must be loaded in as they come into view, and many elements must be unloaded as they go out of view. This technique is also referred to as streaming.

Entities are far more suited for streaming than GameObjects because entities consume less memory and processing overhead, and they can be serialized and deserialized much more efficiently.

Entities Graphics

The Entities Graphics package provides components and systems for rendering entities via the Universal Render Pipeline (URP) or the High Definition Render Pipeline (HDRP). Entities Graphics is built around the BatchRendererGroup API.

The Entities Graphics samples demonstrate various graphics features, such as light probes and lightmaps, material property overrides, and LODs.



An Entities.Graphics sample scene in the EntityComponentSystemSamples repo

Physics

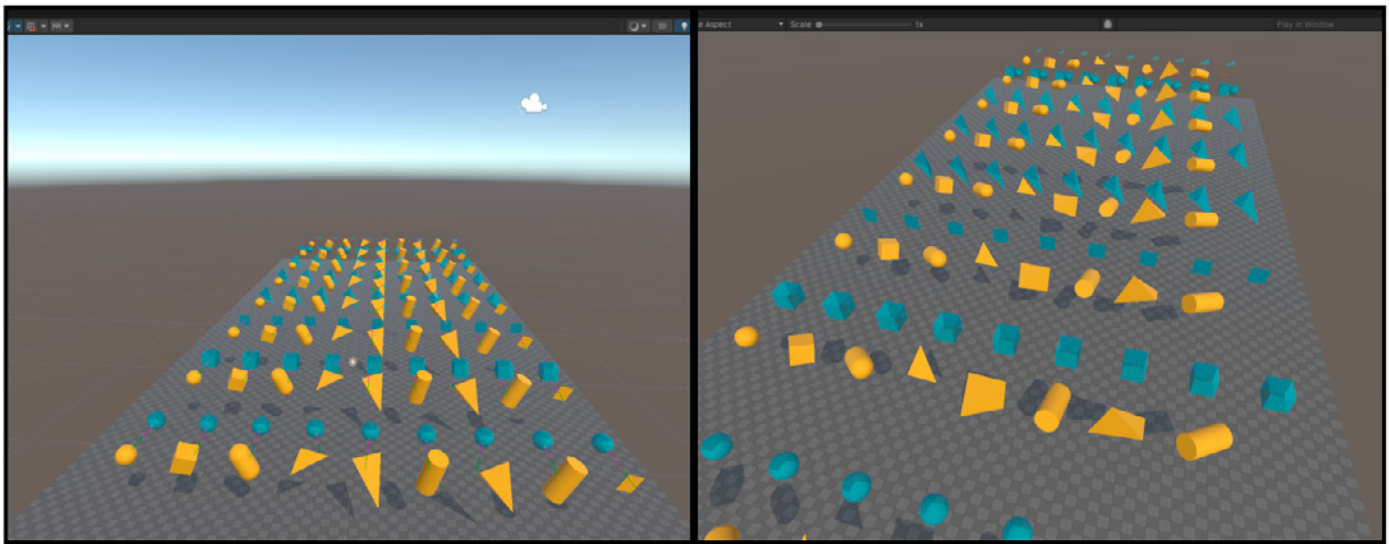
The Unity Physics package provides rigid body simulation and collision checks.

Unity Physics supports swapping in alternate “backends” while maintaining the same surface level API, allowing you to swap physics implementations without changing your own code or assets.

The default backend provided in the package is deterministic, meaning that given the same initial conditions and inputs, it will produce the same results.

The Havok Physics package provides an alternative backend based on the proprietary Havok Physics engine that powers many industry-leading AAA games.

The Physics samples illustrate many features of the package, including colliders, mass and motion properties, material properties, events, joints and motors, and more.



A Physics sample scene in the EntityComponentSystemSamples repo

Netcode for Entities

The Netcode for Entities package is one of two netcode solutions provided by Unity. Unlike the other solution, Netcode for GameObjects, Netcode for Entities uses an authoritative server and supports client-side prediction, making it better suited for fast-paced competitive games.

Authoritative server

Rather than splitting authority of what is happening in the game across the player machines, an authoritative server runs the full game simulation itself and dictates what is happening in the game: the clients send player input to the server, the server updates the game simulation, and the server sends new snapshots of the game state back to the clients. This is the simplest way to implement networked game logic and the one least prone to exploitation by cheaters.

Client-side prediction

Since it takes time for data sent from the server to reach the clients, the state which the client has received always lags behind the server. For many elements in a game, this lag might be acceptable, but for others, like the player's character, such lag can ruin the feel of the game and make it difficult to play.

Client-side prediction can solve this problem. For designated elements like the player's character, the client will attempt to predict the state a fraction of a second into the future. As long as these predictions match the state on the server accurately and consistently enough, the game will feel much more like a zero-lag experience.

On top of these two core features, Netcode for Entities can also scale better than Netcode for GameObjects and provides better means to optimize bandwidth.

A good introduction to Netcode for Entities is the Netcode for Entities samples repo. These samples demonstrate many basic and advanced features, including syncing, connection flows, integration with Unity Physics, and more. Start with the Networked Cube tutorial, which covers:

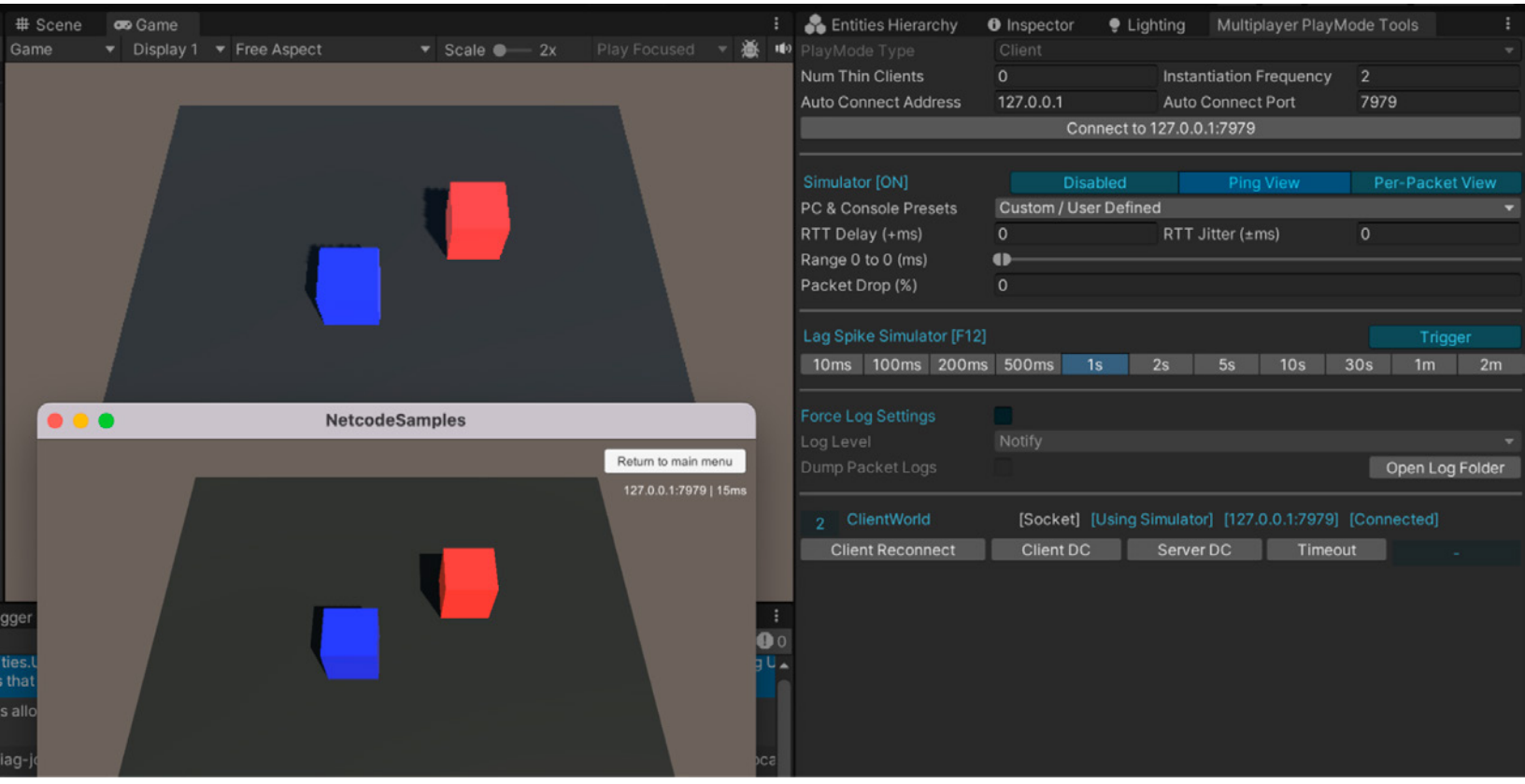
— Establishing a connection with the server

— Communicating with the server

— Spawning synchronized entities on the server

— Creating standalone builds of the server and client

— Running the server and a client in Play mode within the Editor



The Networked Cube tutorial running in the Editor and as a standalone build

ECS Network Racing sample

The ECS Network Racing Sample features car-racing multiplayer mechanics.

The ECS Network Racing sample is a lobby-based multiplayer car racing sample featuring Unity Physics and Vivox voice chat.

Character Controller

The character controller package is available in the Unity Asset Store.

The CharacterController package provides an ECS-based implementation of first- and third-person character controllers that work with Unity Physics and Netcode for Entities. The controllers support various common character behaviors, like sprinting and double jumping. You can also try the CharacterController tutorial and samples for more learning examples.

Animation

Unity is developing a new skinned mesh animation system that will work directly with entities, but as of this writing (Spring 2025), it's not yet available.

In the meantime, the most common solution for animated characters in an ECS-based project is to use animated GameObjects whose transforms and animation states are synced from entities. In other words, the game simulation is fully implemented in entities, but presentation of the animated characters is done with GameObjects. This solution does require some extra coding and induces some overhead, but it should suffice for most games. For a simple demonstration, see the "AnimateGameObject" sample in the sample repo.

Alternatively, some game makers and asset developers have implemented their own custom animation solutions. The community offers several different solutions, some of which are available on the Unity Asset Store.

User Interfaces

Unity does not provide an ECS-based UI system, but ECS-based games can use the GameObject-based UI Toolkit. For a demonstration of how to coordinate between UI Toolkit and ECS, see the DOTS UI sample.

To learn more about UI Toolkit itself, see the following resources:

— UI Toolkit for advanced Unity developers (Unity 6 edition)

— UI Toolkit Sample – Dragon Crashers

— QuizU – A UI Toolkit sample

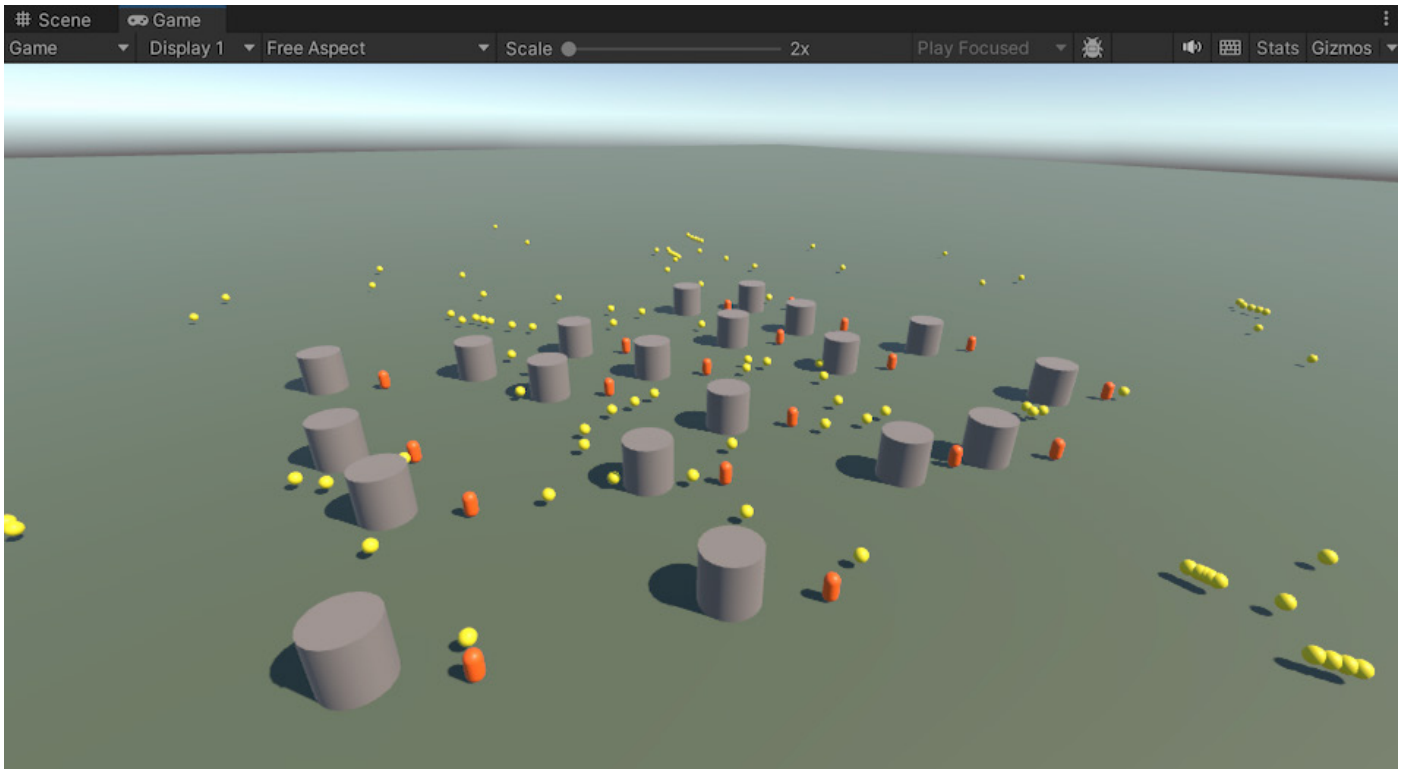
— QuizU article series on Discussions

DOTS educational content

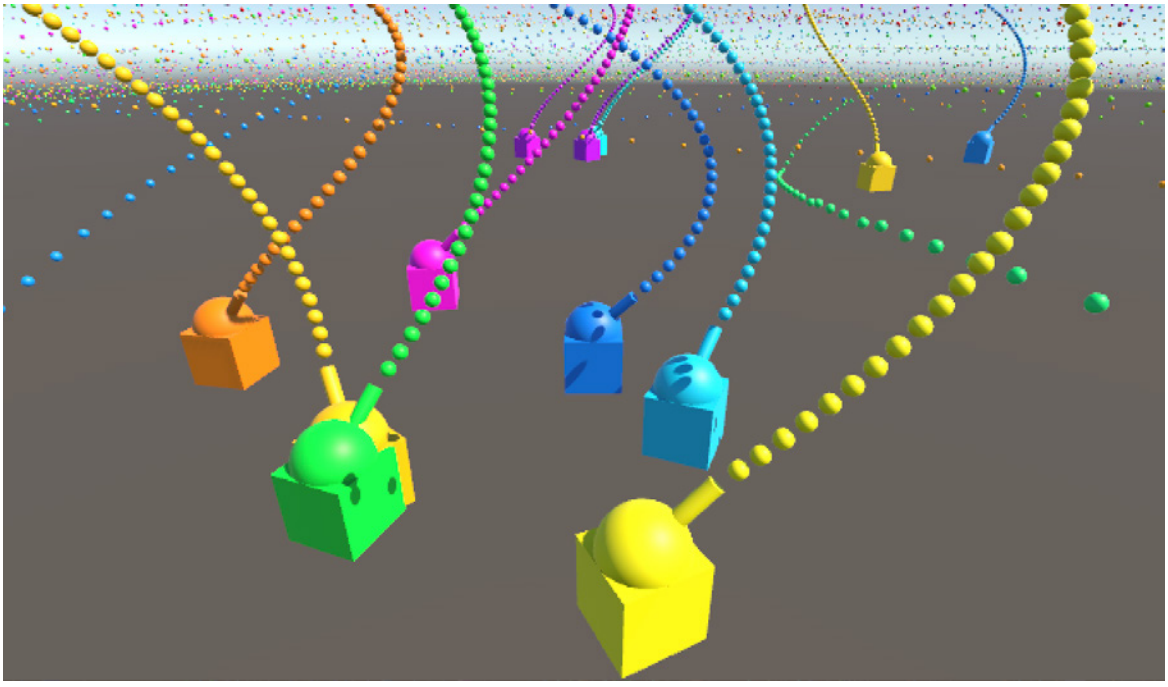
The primary DOTS introductory content is the “DOTS 101” series of samples hosted in this GitHub repo. These simplified samples will introduce you to the functionality in the DOTS packages. For this reason learners are encouraged to go through the series in the listed order:

1. The Job System 101
2. Entities 101
3. Netcode 101
4. Physics 101

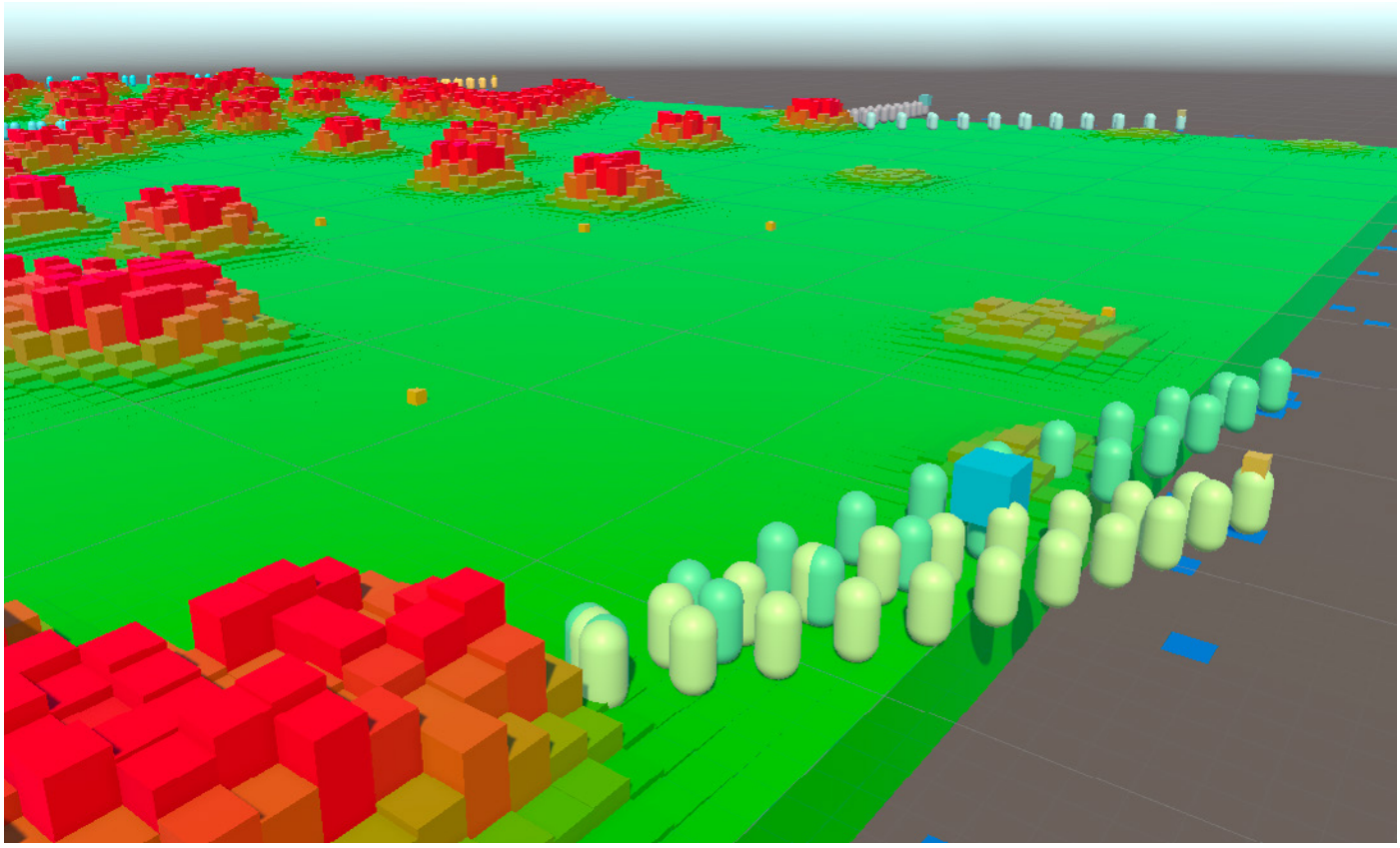
In addition to the DOTS 101 series, the repo also contains many additional samples illustrating intermediate and advanced features and use cases for Entities, Netcode, Physics, and Entities Graphics.



The Kickball tutorial in the DOTS 101 series



The Tanks tutorial in the DOTS 101 series



The Firefighters sample in the DOTS 101 series

Evaluating DOTS for your project

For existing projects

If you have code which is causing CPU bottlenecks, you should consider reimplementing it as Burst-compiled jobs. Not only will Burst-compiled code often run multiple times faster than the Mono- or even IL2CPP-compiled equivalent, jobs allow you to split your workloads across all cores of the CPU.

The good news is that Burst-compiled jobs can usually be integrated into the majority of existing projects with relative ease, even if a project otherwise makes no use of DOTS. Aside from possibly having to copy data into and out of unmanaged collections, rewriting existing code as Burst-compiled jobs generally requires no significant code restructuring.

This is less true for the Entities package: while it's sometimes possible to selectively integrate entities for implementing specific features, ECS architecture tends to impose its own code structure on the whole project.

For new projects

When building a new project, here are four good reasons you might want use Entities:

— The project will have many static elements, such as for rendering a large, detailed environment. The original Megacity project demonstrates a complex environment made out of entities.

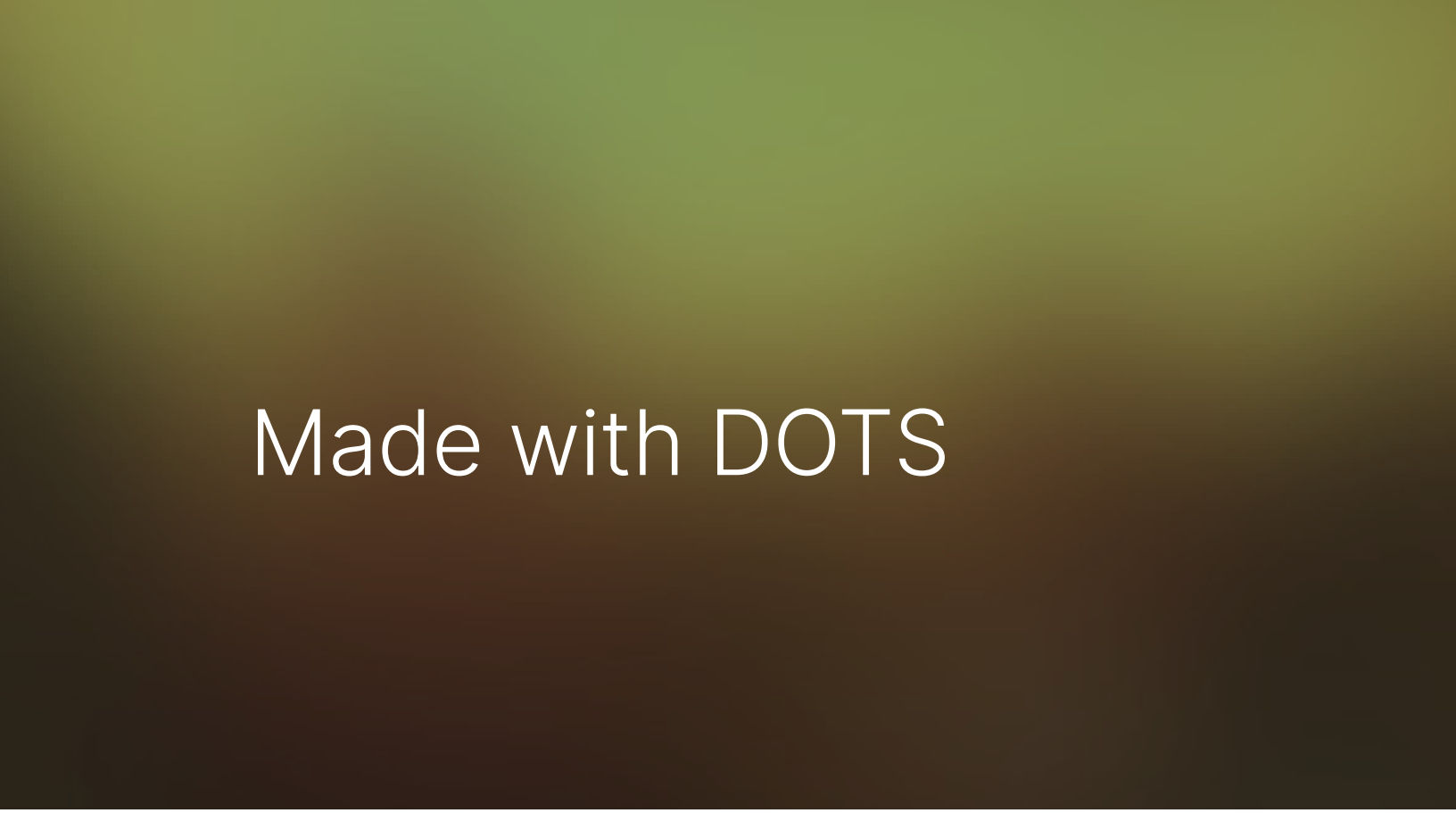
— The project will have many dynamic elements with computationally heavy behaviors. A real-time strategy game, for example, often needs to compute pathfinding for hundreds or thousands of units.

— You prefer the ECS way of structuring data and code, which is arguably easier to reason about and maintain than the more common object-oriented alternative. At the very least, ECS generally makes it easier to profile and identify bottlenecks.

— The project is a competitive multiplayer game with fast action, such as a shooter, and requires authoritative servers and client-side prediction for a good player experience (as stated above, these features are supported in Netcode for Entities but not Netcode for GameObjects).

On the other hand, many games are bottlenecked primarily on the GPU, in which case Entities and the rest of the DOTS packages and related technologies might not help much because DOTS only improves CPU efficiency. Still, if DOTS can help you do the same amount of work in less CPU time, that leaves more headroom for additional features, and extra headroom can also help greatly if you later decide to target lower-powered devices.

The following section highlights some of the games that use Unity ECS and DOTS technologies.



Made with DOTS

Over the last several years, development teams have seen their multiplatform games benefit from using DOTS packages and technologies. As you'll learn from the following excerpts from customer stories, each team carefully thought through how their game could benefit from DOTS before deciding to implement it.

You'll find more Unity customer stories and profiles in the Unity Resources hub.

Made with DOTS: *Bare Butt Boxing*, by Tuatara Games

Bare Butt Boxing by Tuatara Games, made with Unity, available for PC and console

Tuatara Games built Bare Butt Boxing using Unity's DOTS from the very start of development. "Since this is our first game as a new team, we wanted to do early access with a foundation strong enough for us to pivot the design into the right direction," says software engineer Hendrik du Toit. "DOTS allowed us to modularize our systems in a way that we can test gameplay ideas without weeks of rewriting code."

Tuatara's data-oriented design approach simplifies iteration and allows them to be flexible with optimization. "Having ECS means we can adjust runtime data layout easily without impacting serialized data," says game programmer Ewan Argouse.

"ECS has helped us to divide the game into multiple layers without trouble. The game design can be simple and related to the simulation directly, and we can create systems on top of that to present it nicely...thanks to that, the presentation can be complex while our simulation can be client-predicted without being too heavy on the CPU."

- Ewan Argouse, game programmer, Tuatara Games

Made with DOTS: *Histera*, by StickyLock Games

Histera by StickyLock Games, made with Unity, and available for PC via Steam Early Access

"Histera started out as an extraordinary idea, and that was to innovate the FPS genre as a whole. How we've done that is...through the 'glitch'...the glitch is our main USP. (It) basically takes a section of the map and changes that to a completely new era. So you can go from a prehistoric era up to a future era and it will completely change the layout as well.

The reason we chose to go with DOTS was that at the time, we were looking at networking solutions, however there weren't a lot of options for us. Since we're going with a first-person shooter game, we knew that peer-to-peer wouldn't be the reliable option for us. We wanted a dedicated game server option. There was a Unity blog detailing the release of Unity Netcode and DOTS...they also released a few samples. And when we looked into those, we got quite intrigued because they showcased that we would be able to make an FPS with the tech.

Once we dove deeper into those (DOTS) packages, we found that it was very interesting for us from a developer perspective...it was a completely new paradigm. Instead of being object-oriented, it was data-oriented...after a lot of talking and discussing, we wanted to push forward and invested in our knowledge on DOTS and ECS."

- Jamel Ziaty, producer, StickyLock Games

Made with DOTS: *V Rising*, by Stunlock Studios

V Rising by Stunlock Studios, made with Unity, available for PC

When Stunlock Studios set out to build *V Rising*, they realized quickly that the scale of their vision would require a different design pattern from their previous titles. “We wanted the world to feel alive with lots of destructibles and interactables,” says cofounder and technical director Rasmus Höök.

Höök started experimenting with DOTS “because its use cases seemed to fit perfectly with the problems we were trying to solve.” Using DOTS and ECS, Stunlock decreased server strain and minimized client CPU resources, resulting in more concurrent players, lowered system requirements, and a robust tech stack that can scale up to meet Stunlock Studios’s creative vision.

“ECS offers a significant advantage with its clear separation between Editor data and runtime data. When working in the Editor, we create authoring prefabs, which are essentially standard GameObjects with MonoBehaviors. However, these prefabs are solely for editing purposes and are not directly used in the game itself. Instead, they go through a process called baking where they are converted into runtime components. Since the authoring prefabs are only used in the Editor it allows us to add functionality and data to them to improve the workflow without worrying about cluttering the actual game.

Because of this, we can freely modify and optimize the runtime components without impacting the Editor data. This separation has greatly helped us maintain a complex project like *V Rising*.”

– Rasmus Höök, technical director, Stunlock Studios

Made with DOTS: *Zenith: The Last City*, by Ramen VR

Zenith: The Last City, by Ramen VR, made with Unity, available on PC and console VR

As systems-based games, MMOs require strong, scalable technical foundations. Early in development, Ramen VR organized Zenith's systems using MonoBehaviors, but running logic hundreds of times across hundreds of identical GameObjects was inefficient.

They leveraged Unity's ECS framework to avoid the drawbacks of object-oriented programming. "An MMO is a great application for ECS," [CTO Lauren] Frazier notes. "Zenith requires thousands of Entities to coexist at the same time, and ECS allows us to run at scale."

In the new workflow, every "actor" GameObject (players, mobs, collectibles) has a corresponding ECS Entity. The ECS runs through GameObjects and checks for relevant tags, triggering logic whenever they're found.

"It was nice to be able to pick the workflow appropriate to the situation. We could have done pure Objects or pure Entities – but I don't think you should have to choose," says Frazier."

– Lauren Frazier, CTO, Ramen VR

Made with DOTS: *Den of Wolves*, by 10 Chambers

Den of Wolves, by 10 Chambers, made with Unity, for PC, release date to be announced

The upcoming *Den of Wolves*, from 10 Chambers, is a four-player co-op fps heist game set in the futuristic Midway City, an unregulated innovation zone in the Pacific. Players operate as criminals for hire in the conflicts between rival corporations.

10 Chambers has been using Unity for a number of years, including for their previous hit game *GTFO*. They upgraded to Unity 6 and DOTS soon after starting production on *Den of Wolves*.

“We like to build a lot of our tools ourselves, but when the team at Unity showed us a scene with our assets in Unity 6, we were blown away…we’re fans of fast running optimized games and the extensibility and scalability of DOTS helps us make sure that *Den of Wolves* runs smoothly.”

– Simon Viklund, co-founder and audio and narrative director, 10 Chambers (from the Unite Barcelona 2024 keynote)

"We intended for *Den of Wolves* to be made with Unity 2022 LTS but after seeing the improvements in Unity 6, both in terms of performance and rendering quality, upgrading became obvious…We wanted to move as much heavy lifting as possible from the CPU to the GPU, and make everything else on the CPU as performant as we could. The first part means HDRP and the second part means DOTS."

– Svante Vinternatt, co-founder and COO, 10 Chambers (from the Unite Barcelona 2024 keynote)

"The switch from object-oriented to data-oriented (design) is one that gives us more structure (and) much better separation between data and execution.

It's quite a big undertaking for our programmers…There are a lot of things to learn to really get ECS up and running, to get your systems and data right. So you need to take some care in how you design it, you need to care about the architecture.

But once you do that, you have a great foundation for maintaining your game for a long time. It's…important to know that it *is* an investment, in skill and time, in the beginning. But it's one of the benefits with Unity and now we're not limited by (the) technical boundaries that we had before."

– Hjalmar Vikström, co-founder and chief development officer, 10 Chambers (from *Den of Wolves*: Building a Co-Op Shooter Game in Unity 6)

Made with DOTS: Megacity Metro sample

The Megacity Metro sample

Unity's Megacity Metro sample is a multiplayer and mobile-focused spin on the original Megacity sample. Megacity Metro is built with URP, Entities, Netcode for Entities, and Unity Physics, and runs on a range of devices, from low-end mobile to high-end console platforms. It supports competitive cross play for over 100 players and showcases Unity Gaming Services like Authentication, Game Server Hosting, Matchmaker, and Vivox.

You can learn more about Megacity Metro and download the project here.

Appendix I: Misconceptions about DOTS and Unity Entities

False: DOTS, ECS, Unity Entities, and data-oriented design are all the same thing

These are all distinct and different things:

- **Entity Component System (ECS)**: This is an architectural pattern that originated in some games from around the early 2000s. There are at least several variants of this architecture, but they are all built around two common core ideas:
  - A clear separation of data and code (entities and components are the data; systems are the code)
  - A composable data structure (entities composed of components)
- **Unity Entities**: This is a Unity package that provides a variant of ECS architecture. You'll often see "ECS" or "Unity ECS" used as synonyms for Unity Entities.
- **Data-oriented design (DoD)**: This is a set of design principles about performance and problem solving that also emerged in the 2000s, partly in conjunction with ECS. Not to be confused with "data-oriented programming", "data-driven design", or "data-driven programming", which are separate concepts despite the similar names.
- **Unity's Data-Oriented Technology Stack (DOTS)**: This is a set of Unity features built on DoD principles and which includes:

- The Entities package
- The Collections package
- The Mathematics package
- The Burst compiler package
- The C# job system
- It also includes these feature packages built on top of the above:
  - The Entities Graphics package
  - The Unity Physics package
  - The Netcode for Entities package

False: Using DOTS requires using entities

While some parts of DOTS depend upon entities (Entities Graphics, Unity Physics, and Netcode for Entities), the other parts do not. In fact, games built just with GameObjects but no entities can often make effective use of Burst-compiled jobs along with the Mathematics and Collections packages.

Not only can Burst-compiled jobs often alleviate key bottlenecks, they are usually relatively painless to retrofit into an existing codebase. In contrast, retrofitting code built around GameObjects to use Entities commonly requires an invasive rewrite.

Note: Currently, work is under way to improve interop and integration between Entities and GameObjects, which should make Entities much easier to selectively retrofit late into a project. However, these features won't ship in Unity 6.x.

False: Using DOTS will make any Unity game significantly faster

Even among games with CPU bottlenecks, many games suffer from inefficiencies that DOTS does not directly address. For example, it's quite common for games to get bogged down by heavy-weight UI, thanks to slow UI layout or excessive overdraw. In such cases, the whole game might run slow even if the core game simulation is perfectly adequate to hit the game's performance targets. The only solution, then, is to fix the parts of your game that are actually too slow!

On the other hand, many games end up with excess costs spread throughout their code and each frame, such that it's hard to identify clear opportunities for optimization. This kind of 'death by a thousand cuts' easily arises in a codebase where performance was not a key concern from the start and closely monitored throughout development. Building a project with DOTS can't guarantee that you'll avoid this pitfall, but it can greatly help.

False: Every new Unity project should use entities

As discussed just above, there are many games where the potential performance benefits of entities (or DOTS more broadly) won't necessarily make a big difference. Maybe your game is really only GPU-limited, or maybe, on the whole, your game just isn't that demanding for your target hardware. Also as mentioned above, Burst-compiled jobs are easier than entities to retrofit late into your project.

The hard question, then, is whether to build the foundation of your game on entities. In short, there are five major reasons to do so:

— **Your game will have an ambitious scale simulation.**

— **Your game will have ambitious scale environments.**

— **Your game needs netcode with client-side prediction.**

— **You want to use Unity Physics.**

— **You prefer ECS as a way of writing code** (see the next section).

Note: Currently, only Netcode for Entities provides client-side prediction, which is important for certain kinds of fast-action netcode multiplayer games. However, client-side prediction for GameObjects is a feature currently in development, as part of a broader long-term initiative to unify Netcode for GameObjects and Netcode for Entities.

False: The benefits of ECS are just about performance

Though most programmers today are familiar with object-oriented programming, the procedural style of ECS often makes code and data easier to design and reason about.

For elaboration, see the appendix sections called "Data-oriented design", "Costs of OOP", and "Structural costs of OOP".

False: The performance benefit of entities/ECS is all about memory efficiency and cache utilization

Memory efficiency is a key consideration for high-performance code, and good cache utilization is an important part of that: The more the CPU has to go out to main memory instead of finding what it needs in the cache, the more cycles it's going to spend waiting, instead of doing work.

Unity entities stores entity components in blocks of memory called "chunks", which are organized into sets called "archetypes" (details here). This layout lends itself to good memory efficiency and cache utilization, but Unity Entities also has other performance advantages over GameObjects and typical OOP code:

— Instantiation and destruction of entities is cheaper than for GameObjects: Entities have lower memory footprints and require less bookkeeping, and so can be created and destroyed more efficiently, especially *en masse*. Unlike with GameObjects, pooling entities is never really necessary.

— Whereas GameObjects and their components are always managed objects, entities and their components are unmanaged (meaning they are manually allocated rather than managed by the garbage collector). So entities not only minimize allocation costs, they avoid garbage collection overhead.

— Because entity components are unmanaged, they can be accessed in Burst-compiled code and jobs, which often provide enormous performance gains.

— Entity queries allow for easy, fast processing *en masse*. They help avoid the one-at-a-time anti-pattern common in OOP that can create bottlenecks and be difficult to optimize.

So yes, the potential performance gains of entities are partly about better cache utilization, but cache is far from the only performance consideration. In fact, for many typical game scenarios that involve only dozens or hundreds of things to process (rather than thousands or millions), these other factors are often more significant.

False: Entities are the ultimate, optimal data structure for everything

A core principle of data-oriented design is that different problems require different solutions. The corollary of this is that overly abstract, generalized solutions are sub-optimal, especially for performance.

Entities, in a sense, are a very generic data structure, and despite being flexible, they cannot be optimal for all possible use cases:

— While hierarchical relationships, such as in transform hierarchies, *can* be expressed with entities, doing so is neither convenient nor particularly efficient.

— Similarly, ordered relationships between entities cannot be conveniently or efficiently expressed.

— While entities are much more compact and cheaper to instantiate than GameObjects, it wouldn't make sense to represent, for example, every vertex of a mesh as an individual entity, just as it wouldn't make sense to represent them as individual GameObjects. The only sensible way to store mesh vertices is in tightly-packed arrays, and the same is true for many other kinds of data.

So the best way to think of entities is that they are a good *default*, but not ideal for everything.

In the context of a general-purpose game engine:

— Entities can serve as a common data protocol between various parts of the engine.

— Entities are uniformly inspectable and editable in the editor.

— Entities lend themselves to easy, fast serialization.

In your own game code, you generally won't go too wrong modeling your data as entities. On the other hand, you should keep an eye out for cases that warrant just simple arrays or lists. Likewise, you should consider using trees, hashmaps, lists, graphs, or other more specialized structures for data with complex inter-relationships.

Keep in mind that these alternative data structures can always be stored indirectly in components of your entities. For example:

— A tree can be stored as a BlobAsset that is referenced from an entity.

— An array or list can be stored as a DynamicBuffer component of an entity.

— A managed C# object can be referenced from an entity as a UnityObjectRef field of a component.

Referencing all of your data structures from entities in this way will generally make them easier to work with in your ECS code. For example, instead of having to somehow pass a hashmap around to your systems, your systems can just grab the reference to the hashmap from a queried entity.

False: Multithreaded programming is too hard for most programmers

Multithreaded programming is an intimidating topic, and even experienced programmers struggle to get it right. There are two main challenges with it:

— **Thread synchronization:** The access of data and other resources which are shared between threads must be carefully coordinated, usually by means of spinlocks, semaphores, and other synchronization primitives. Failure to do this coordination correctly can produce race conditions, deadlocks, and other bugs.

— **Thread management:** The threads themselves must be created and destroyed, and work must be farmed out to the threads. Failure to do this intelligently can undermine the potential performance gains of multithreading.

Unity's job system greatly simplifies these concerns: the job system itself is responsible for managing threads, and the job safety checks allow even inexperienced programmers to easily write correct, multithreaded code without manual synchronization.

False: Manual memory management is too hard for most programmers

If, like many programmers these days, you only have experience in garbage collected languages, having to manually manage memory may seem like a daunting prospect. This fear, however, is mostly misplaced, as manual memory management is very tractable once you get the hang of a few key patterns. Conveniently, DOTS provides a safe environment for you to code without a garbage collector for the first time:

For starters, when using Unity Entities, the entities are manually allocated, but the allocations and deallocations are handled for you when you create entities and destroy entities or add components and remove components. You are still responsible for destroying the entities which are no longer needed, but generally this is a natural part of your game logic. Although keeping entities alive too long technically counts as a memory leak, in practice, such mistakes manifest as bugs in your game logic, so they tend to be very visible and relatively easy to track down.

Aside from entities, DOTS also provides unmanaged collection types and a set of arena allocators. For a large majority of cases, proper allocation and deallocation of these collections is handled by just picking the right allocator. Does the data in question need to live for the full duration of the game? Use the Persistent allocator. Does the data just need to live for the duration of the current frame? Then use the Temp or WorldUpdate allocators.

Lastly, the DOTS disposal safety checks will catch most cases where you fail to properly allocate or deallocate a collection, so even when you make a mistake, the source of the problem is usually very easy to track down and rectify.

False: GameObjects and Entities cannot be used together, and (Mostly) False: Entities cannot animate, emit sound, or do UI, etc.

Adding Unity Entities or any other DOTS package to a project does nothing to change how GameObjects and other Unity features operate, and in fact, most “entities/ECS projects” will rely upon GameObjects for various aspects of Unity functionality. For example, Unity’s UI solutions all depend upon GameObjects, so a project using entities with UI will still need some GameObjects.

There is currently no Unity-provided solution for directly animating entities or having them emit sounds, but the desired end result can be achieved indirectly by coordinating your entities with GameObjects. For example, an animated character can be represented in your simulation as an entity that has an associated GameObject which does the actual animation and rendering. This requires you to sync the transform and animation state between the entity and GameObject, which incurs some hassle and overhead, but the cost is usually not a big deal as long as you don’t have hundreds or thousands of characters.

Note: A new Unity animation system is in development which will support both entities and GameObjects. With this new system, entities will be directly animatable without use of GameObjects.

More generally, the use of some GameObjects isn't necessarily a notable performance problem as long as the GameObjects stay numbered in the tens or hundreds rather than thousands or beyond, and as long as the computationally heavy lifting is handled by entities and Burst-compiled jobs. Typically in an entities project, you should prefer using entities for your core simulation logic, physics, and most rendering, but then use GameObjects for a few parts of presentation, such as animation, sound, and UI.

(Mostly) False: DOTS code cannot use managed objects

As described in the previous section, there are cases where you'll need to coordinate between entities and GameObjects or other managed objects. However, the key restriction of jobs and Burst-compiled code is that they cannot access managed objects. The consequence of this is that coordination of GameObjects and entities requires either:

— Accessing entities from GameObject code

— Accessing GameObjects from an ECS system that isn't Burst-compiled

Generally, accessing entities from GameObject code is discouraged because GameObject updates do not coordinate with the ECS job safety checks, and besides, the alternative is generally a cleaner code pattern.

Note: Technically, a managed object *can* be accessed in a job, but doing so is only safe if you "pin" the object for the duration of the job. Doing this is, generally, not worth the hassle, as the whole point of jobs is to maximize CPU performance, which means you basically always want your jobs to be Burst-compiled. Because Burst-compiled code cannot access managed objects, there's rarely a good reason to access managed objects in a job.

Another note: For storing managed objects in entity components, you should use UnityObjectRef (another option is to use managed components, but these are less efficient and may be phased out in future versions).

Appendix II: Hardware concepts related to performance

What follows is a brief discussion of hardware concepts and how they relate to performance. Understanding these concepts can help you understand the design choices behind DOTS and how to make good performance choices in your own code.

Memory allocation and garbage collection

In modern operating systems, programs run as separate processes, where the memory of each process is managed by the operating system. When a process wants more memory, it has to request it from the operating system, upon which the operating system will give the process a contiguous block of memory. This is called memory allocation.

When a process is terminated, the operating system will reclaim the memory, freeing it up to be used elsewhere. For long-running programs, however, it often makes sense for the program to hand back blocks of memory which it is no longer using. This is called memory deallocation or freeing. In simple short-lived programs, it's often sufficient to only allocate memory without ever freeing it. However, if a long-running program continues to make new allocations but neglects to ever free them, the program might end up using an unreasonable amount of memory. These situations are called memory leaks, and can lead to worse performance and instability.

Programs often use their own internal allocators, which can work in the following way:

1. The program allocates a large block of memory from the operating system.
2. The program's own internal allocator tracks which ranges within the block are currently in use.
3. When the program needs more memory, it requests it from the internal allocator rather than from the operating system.
4. When these internally allocated blocks of memory are no longer needed, the program should notify its allocator to free the memory.

Internal allocators have some advantages.

— Unlike allocating and deallocating from the operating system, allocating and deallocating from an internal allocator does not generally require expensive system calls.

— A program can use multiple custom allocators to better accommodate different use cases: Some allocators are more appropriate for small, short-lived allocations, while other allocators are more appropriate for large, long-lived allocations. For example, a so-called "arena allocator" frees all of its allocations at the same time, so its internal logic and bookkeeping can be very simple and cheap.

In many of today's popular languages, including C#, the runtime uses a garbage collector that will scan memory to find unused allocations and free them. Compared to manual allocation, this automated way is more convenient and makes memory leaks and other memory-related issues easier to avoid. On the downside, garbage collection incurs overhead and requires interrupting the program execution, which may cause noticeable pauses that negatively affect the player experience.

In DOTS, entities and the native collections are unmanaged, meaning they are not allocated or managed by the runtime or its garbage collector:

— For entities, the memory is allocated and freed for you by the EntityManager, so your entities only leak memory if you neglect to destroy them when they are no longer needed. In practice, such cases tend to be noticeable, so this mistake is easy to detect and correct.

— For the native collections, DOTS provides several allocators with different trade-offs. The **Allocator.Temp** allocator, for example, provides very cheap allocations that are disposed of automatically at the end of the frame or the job in which it was allocated. The **Allocator.Persistent** allocator, in contrast, provides more expensive allocations that live indefinitely until manually freed. Unlike allocations from Allocator.Temp, allocations from Allocator.Persistent can be large, and passed into jobs. Other allocators include **Allocator.TempJob** and the **WorldUpdateAllocator**.

See the documentation on Unity's garbage collector for more information.

Multithreaded programming

Most modern CPUs have more than one core, and putting these additional cores to work can greatly boost performance of a game that is CPU heavy. However, multithreading can be difficult and unsafe because it requires a lot of low-level manual programming that might be unfamiliar territory for many C# developers. In DOTS, the C# job system helps you write safe multithreaded code in a way that is easy and that avoids common pitfalls, but this section should help you understand the underlying issues.

When a process is spawned by the operating system, it starts off with a single thread of execution. Through system calls, a process can spawn additional threads which will belong to the same process and thus share the same memory.

Each logical core of the CPU can run one thread at a time, and the operating system controls which threads run on which cores and when. At any time, the operating system can interrupt a running thread and suspend it so that another thread can use the core. When two threads access the same resource (namely data), the danger is that one thread mutates the resource when the other thread is not expecting it to. In general, a thread should only read and modify a shared resource within a “critical section” – a span of code within which the thread has exclusive access to the shared resource.

To control access across threads, a shared resource is governed by some kind of synchronization primitive, such as a mutex. However, these synchronization primitives generally require all threads that use them to follow a strict protocol, and failing to do so can make the primitives ineffective or hang the program.

Another issue is that certain system calls may block the calling thread, meaning suspend its execution. For example, when a thread invokes a system call to read a file, the data probably isn’t yet sitting in memory, so it must be copied off a device into memory first before the system call can return. Because the wait for the data may be very long (in CPU terms), the operating system may block the calling thread while the data is being loaded, and another thread can run on the CPU core in the meantime. Only once the data is ready will the operating system unblock the thread and allow it to resume execution.

One use case, then, for multithreading is to do longer-running blocking operations on “background threads”, such as reading files and writing files, while continuing to do work on the “main thread”. An interactive program, for example, may want to load a file in the background while the main thread still responds to user input and redraws the screen.

In other cases, you may simply want to split a program’s CPU workload across multiple cores to get the work done faster. Data compression, for example, is very CPU intensive, so it can usually benefit from multithreading.

A consideration to keep in mind is that the CPU cores must contend with each other for use of the storage devices, system memory, and other system resources. For example, if two threads try to concurrently access memory, they cannot do so at the same moment but instead must take turns. Fortunately, these overlaps are resolved at the hardware level, but the problem

remains that each thread's memory access slows down the overlapping memory access of all other threads. Particularly then, for a task which requires heavy amounts of memory access relative to CPU computation, splitting the work across multiple threads will tend to reap diminishing returns.

For example, you might assume that a task split across 10 threads should run 10 times faster than the same task running on just one thread, but this theoretical limit is rarely achieved in practice, thanks to memory contention. Instead, 10 threads might more realistically get you perhaps a 5-7x boost (though, again, this depends upon the specific scenario). In fact, if the task requires a high ratio of memory access relative to computation work, the performance might improve by using *fewer* threads, because the fewer the number of threads the less memory contention.

Memory and CPU cache

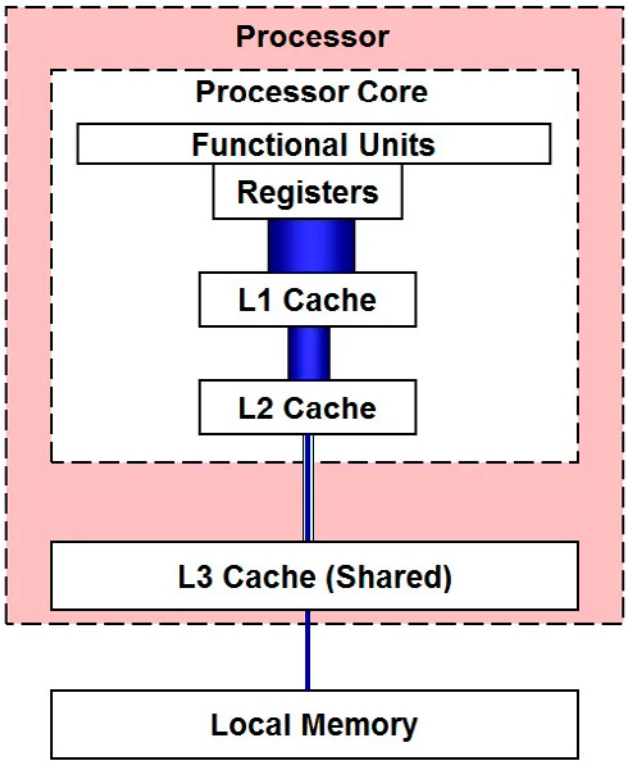
Most of today's CPUs have one to three levels of cache, usually designated L1, L2, and L3.

When the CPU executes an instruction that reads an address of system memory, the hardware first checks if a copy of the data at that address is sitting in the L1 cache. If not, the hardware then checks L2 cache (if it exists), and then L3 cache (if it exists). If no copy currently sits in any cache the hardware will actually read system memory itself. Upon reading the data, the hardware will copy it into a portion of the lower cache levels, e.g., data read from L3 will be copied into L1 and L2, and data read from system memory will be copied into all cache levels. This caching strategy makes sense because when an address is read, it's usually likely that the same address will be read again shortly thereafter. By copying the data into cache, the data can likely be read directly from cache the next time it is needed.

Of course, not all of system memory can fit in cache: Each level of cache is smaller than the one above it, and the largest cache is still much smaller than all of system memory. Therefore, when a portion of memory is copied into cache, it must overwrite some other portion of memory that was previously cached.

When data is read from cache, it's called a "cache hit". When data must be read from system memory itself because a copy is not currently cached, it's called a cache miss.

The precise performance characteristics of caches varies greatly among different chips but roughly, L1 will be at least a few times faster than L2, L2 will be at least an order of magnitude faster than L3, and L3 will be at least twice as fast as system memory. In total, the CPU will likely access data in L1 cache at least two orders of magnitude faster than it will access data in system memory. Because the speed gap between lower cache levels and system memory is so great, minimizing the number of cache misses triggered in your program is a key performance consideration.



Levels of cache; Source: https://tech4gamers.com

Thanks to a hardware feature called prefetching, the simplest and most effective way to minimize cache misses is to access addresses of memory sequentially rather than jumping around randomly. Therefore, cache-efficient data structures store their elements tightly-packed and contiguous in memory. In other words, the data is stored in arrays.

When you start reading memory addresses in sequence, the hardware notices this pattern and will start reading ahead and copying the memory into cache, in the expectation that you'll keep going. This may end up as a bit of wasted effort in cases where the extra data isn't needed, but in cases where you are reading through an array, this prefetching behavior will put data in cache right before the CPU needs it. So aside from a possible initial cache miss when you read the first bytes of an array, an array can be read without triggering cache misses. As the CPU train speeds along, the tracks are laid right in front of the train, just in time (see a dramatization here).

Managed C# objects like GameObjects and MonoBehaviours are separately instantiated, and therefore might end up stored in different parts of memory. Consequently, traversing through many managed objects typically requires jumping around memory and thus triggering many cache misses.

In DOTS, entities and their components are tightly packed in contiguous arrays by design, allowing them to be sequentially traversed with minimal cache misses.

See this talk by Scott Meyers for further information about memory and cache: CPU Caches and Why You Care.

Appendix III: Writing software for performance

Costs of object-oriented programming

A common challenge with object-oriented programming (OOP) is its many definitions. Some insist that OOP is all about inheritance, or polymorphism, or encapsulation, or the combination of the three, while others offer less conventional theories. Here is the definition according the Wikipedia:

> "Object-oriented programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and code: data in the form of fields (often known as attributes or properties), and code in the form of procedures (often known as methods). **In OOP, computer programs are designed by making them out of objects that interact with one another.**" – Wikipedia

In other words, an object-oriented program is composed of interacting "objects", where each object is an encapsulated unit of data and code that has some degree of autonomy and independence from the others. Much like the programs on a network cooperate by sending each other messages, the objects in an object-oriented program cooperate by invoking each others' methods, and in fact, it's the interactions of the objects that really defines object-oriented programming, not the individual objects themselves.

The theoretical benefits of OOP include:

— **Composability**: Programs made out of objects can be incrementally assembled and modified.

— **Reconfigurability**: Features can be easily added, removed, and modified by inserting, removing, and replacing objects.

— **Code reuse**: Objects can be easily reused between programs.

— **Intuitiveness**: Real-world things and processes naturally correspond to objects.

— **Abstraction**: Objects allow the programmer to solve problems at a high-level without being distracted by low-level details.

Steve Jobs elaborated this last point in an interview in the June 16, 1994 edition of *Rolling Stone*:

> "Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.
>
> Here's an example: if I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."
>
> You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level."

Performance costs of OOP

On the downside, OOP tends to incur a number of performance costs:

— **Scattered data layout:** OOP code is often split into many small objects, and the data often ends up scattered throughout memory (which leads to cache inefficiencies, as discussed in prior sections).

— **Excessive abstraction:** Object-oriented design often encourages layers of delegation, where the higher levels defer the real work to lower levels, resulting in many objects and methods that do little actual work.

— **Complex call chains:** Thanks to the many layers of abstraction and a preference for small functions, call chains get very complex.

— **Virtual calls**: Not only do virtual dispatch tables incur overhead over regular function calls, virtual calls cannot normally be inlined (though some JIT compilers may do so at runtime)

— **Bad allocation patterns:** The complex code paths that OOP encourages often make it difficult to reason about object lifetimes, so OOP code tends to rely upon frequent, small allocations and garbage collection rather than more efficient alternatives.

— **One-at-a-time processing:** Because the code which directly manipulates an object is part of the object itself, there's a natural tendency in OOP to process objects one-by-one rather than in large batches.

Structural costs of OOP

Even if we're happy to sacrifice optimal performance for the sake of making programs easier to write and maintain, OOP does not necessarily provide these benefits. Here are a few reasons:

1. Entangling your data and code makes both of them messier and more complicated.

It is often claimed that OOP prioritizes data over code:

> "Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. [...] OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them."
>
> – Alexander S. Gillis, *What is Object-Oriented Programming*, published on TechTarget Network.

However in reality, OOP entangles data and code together: If an object's capabilities must directly follow from its data and vice versa, what an object can do is integral to its definition and can't be separated from the object's data.

This entanglement often leads to questionable design choices:

— Objects with code that really should just have data

— Objects with data that really should just have code

— Objects that group data together for the purposes of code

— Objects that group code together for the purposes of data

— Code that is split across objects for the sake of data

— Data that is split across objects for the sake of code

2. Replacing concentrated complexity with scattered complexity increases the overall complexity.

According to the rules of object-oriented design, an object with too many "responsibilities" should be broken up into smaller objects. However, when you break up large things into smaller pieces, you may end up just scattering the complexity around rather than reducing the overall complexity. In fact, a code base with many small pieces often makes it hard to discern what purpose any one piece of data or code serves and hard to find the parts of code relevant to a given feature.

So while object-oriented design aims to bring clarity to your code as long as you correctly delegate responsibilities amongst a properly designed set of objects, the object-oriented design process can itself often be burdensome and fraught with conjecture, and the typical resulting program structures become excessively fractured.

3. Objects make it difficult to track which code accesses which data

Understanding a program ultimately boils down to understanding its data and how that data gets transformed. The easier it is to reason about the data, the easier it is to reason about the program. Whether adding features or fixing bugs, the programmer needs to be able to determine which code affects a given piece of data and, from the other perspective, which data is affected by a given piece of code.

In an OO program, the more connected the objects, the more difficult it is to make these determinations. Although object encapsulation may keep direct access to a piece of data private, any indirectly connected object may have indirect access through some path of public method calls. For example, when debugging why a value is being incorrectly set, identifying all relevant paths of code may require a lot of detective work. In contrast, in a strictly procedural program, identifying all paths of code that may affect a piece of data usually requires considering many fewer possibilities (as long as the program does not use global variables recklessly).

Data-oriented design

The term data-oriented design (DOD) was coined in the 2000's to describe a set of ideas emerging at the time among some game programmers and others interested in high performance software. No one source has the authoritative definition of data-oriented design, but these resources perhaps come the closest:

— Data-Oriented Design and C++ and Building a Data-Oriented Future: Two talks by Mike Acton

— Data-Oriented Design: A book by Richard Fabian

— Data-Oriented Design Resources: A collection of links about DOD

Here, rather than give a theoretical account, we'll just distill DOD into several points of practical advice:

Design your data before designing your code

The central premise of DOD is that **data is at least as important as code**. At both the macro and micro level, **programs are ultimately about transforming and producing data**, so the nature of your data should dictate the structure of your code rather than the other way around. This is true not just at the beginning of a project but at all stages, so when adding or changing features, you should first reevaluate the structure of your data before restructuring the code.

Note that this conflicts with object-oriented design, where objects inextricably link data and code together. Mixing the design of data with concerns about code complicates the design process and often leads to suboptimal design choices. Conversely, allowing data the freedom to change without immediate concern for code simplifies the design process and typically produces simpler, more optimal data.

Prefer simple data

As a general tendency, simple data leads to simple and efficient code. In particular, you should favor arrays over hierarchical structures and graph structures: Arrays are the simplest way to store many elements of data, and sequentially looping through flat arrays is the most efficient way to access memory.

You should also be careful about creating connections between elements of data (*via* pointers and array indexes) that aren't necessary. Correctly maintaining these connections complicates your code, and traversing connections requires suboptimal random lookups.

Think of your code as a data pipeline

Once you have a rough draft design of your data, the next question is what transformations your data must undergo:

— In a server, client requests and database data are transformed into server responses.

— In a compiler, source code is transformed into machine code or some kind of intermediate code.

— In an audio encoder, audio data of one form is transformed into another.

— In a video game, the user input and game state of one tick is transformed into a new game state, which is then transformed into a new rendered frame.

Of course, these macro-level transformations break down into a number of substeps, but the goal remains the same: For some beginning state of the data, you simply need to connect the dots to reach the expected end state. The code can then be naturally structured as a "data pipeline" in which each step transforms or produces data to be handed to later steps of the pipeline.

This description of programming may sound too simple and obvious, but compared to other theories of how to make software, it offers great clarity. Once you have well-defined start

and end points, figuring out exactly how to get from point A to B is a very concrete, tractable problem, and each separate transformation can be written and rewritten independently from the rest.

This model makes working solutions not only easier to create but also easier to optimize:

First, identifying bottlenecks in a sequential series of steps is as simple as profiling all the steps. A minority of steps will usually account for most of the cost, giving you a clear idea of where optimizations would be most impactful. Therefore, it's important to consider the cost vs gains when prioritizing your optimization efforts and being pragmatic in your performance optimization process.

Second, the pipeline model makes it easier to find optimization opportunities. Very often you will find cases where:

— Certain data should be transformed into an intermediate form that lends itself to more efficient processing by later steps.

— Data that is redundantly produced by multiple steps should instead be cached once in an earlier step.

— Separate steps that access the same data should wholly or partly be consolidated into fewer steps to reduce the overhead of repeated access.

— Some elements of data that are processed one-by-one should instead be processed *en masse*, which generally leads to more efficient memory access, less branching, and less function call overhead, among other efficiencies.

Lastly, a data pipeline lends itself to parallelization: As long as you have clear separation of which steps touch which data, it's easy to identify which steps can be safely processed concurrently.

Measure, estimate, and budget performance at all stages of development

A common mistake in game development is waiting to fix performance at the end of a project. Late optimization work is both costly and risky because:

— Many optimizations are harder to do late in the project.

— Late optimization takes an unpredictable amount of time and effort.

— Late optimization might fail to achieve acceptable results.

Instead of waiting, the healthier practice is to concern yourself with performance from the very beginning of a project. Even if you're willing to tolerate suboptimal performance throughout your prototyping and beta phases, you should at the very least continuously reestimate the needs of your project and establish a performance budget. How much memory, CPU, GPU, storage space, and network bandwidth can you afford for each feature and for the game as a whole? Do the target numbers differ across your target platforms? These are questions that you should reassess at all stages of development.

To learn more about profiling check out our 70+ page *Ultimate guide to profiling Unity games* which brings together advanced knowledge and advice from external and in-house Unity experts on how to profile an application in Unity, manage its memory, and optimize its power consumption from start to finish.

Prefer specific solutions over abstractions

In programming, “abstractions” are generalized solutions that hide internal details behind simplified exteriors. Abstractions come in various forms, including functions, objects, libraries, frameworks, programming languages, and even game engines.

While some degree of abstraction is sensible, excessive enthusiasm for abstraction can cause problems:

— A major reason to use off-the-shelf abstractions, like libraries, frameworks, or game engines, is that they can spare you from difficult and time consuming implementation work. However, a cost of this convenience is often awkward mismatches between the provided solution and your specific needs. Ultimately, bending an off-the-shelf abstraction to your purposes may actually end up being more work than just writing your own specific solution.

— Abstractions often incur many hidden performance costs, like heavy memory footprint or CPU overhead, costs which are paid ultimately for the sake of features you may not even be using.

— In your own implementations, it’s tempting to create an abstract solution that generalizes beyond your current needs in anticipation that it may be useful later in the project. More often than not, however, this kind of speculative work ends up creating more work than it solves, and the resulting solution is often suboptimal and hard to optimize. Abstractions, in fact, may actually end up making changes *more* difficult later when your requirements no longer neatly fit the abstraction.

Instead of worrying about how your requirements might change later, you’re almost always better off solving for your current requirements as you currently understand them. Embrace iteration: You won’t fully understand your problem until you’ve tried to solve it, and you can simply wait to change your code later after your requirements actually change. What they say about writing in general applies to writing code: Good writing is rewriting. What makes code easy to rewrite better than anything else? Simplicity.

If you feel tempted to abstract, the best advice is to wait: solve for at least a few specific cases first, and only then consider combining their solutions into an abstraction. As Richard Fabian writes:

> "Data-oriented design is current. It is not a representation of the history of a problem or a solution that has been brought up to date, nor is it the future, with generic solutions made up to handle whatever will come along. Holding onto the past will interfere with flexibility, and looking to the future is generally fruitless as programmers are not fortune tellers. It's the opinion of the author that future-proof systems rarely are."

In other words, beware: *premature abstraction is the root of all evil*.

More advanced resources from Unity

Find all of Unity's technical e-books in the best practices hub and the best practices section of Unity documentation. You'll also find tech tips in the Technical Articles section of Unity Discussions.


Unity®